Studying the evolution of software through software clustering and concept analysis

Davey, John William

Studying the Evolution of Software through Software Clustering and Concept Analysis

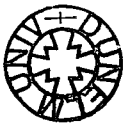
John William Davey

M.Sc. Thesis

Department of Computer Science

University of Durham

September 2001



ABSTRACT

This thesis describes an investigation into the use of software clustering and concept analysis techniques for studying the evolution of software. These techniques produce representations of software systems by clustering similar entities in the system together. The software engineering community has used these techniques for a number of different reasons but this is the first study to investigate their uses for evolution. The representations produced by software clustering and concept analysis techniques can be used to trace changes to a software system over a number of different versions of the system. This information can be used by system maintainers to identify worrying evolutionary trends or assess a proposed change by comparing it to the effects of an earlier, similar change.

The work described here attempts to establish whether the use of software clustering and concept analysis techniques for studying the evolution of software is worth pursuing. Four techniques, chosen based on an extensive literature survey of the field, have been used to create representations of versions of a test software system. These representations have been examined to assess whether any observations about the evolution of the system can be drawn from them. The results are positive and it is thought that evolution of software systems could be studied by using these techniques.

TABLE OF CONTENTS

CHAPTER ONE – INTRODUCTION 6

1.1 PROBLEM 6

1.2 PROPOSED WORK 8

1.3 CRITERIA FOR SUCCESS 8

1.4 DOCUMENT STRUCTURE 9

1.5 CONCLUSION 10

CHAPTER TWO – BACKGROUND AND DEFINITIONS 11

2.1 SOFTWARE EVOLUTION 11

2.2 CLUSTER ANALYSIS AND SOFTWARE CLUSTERING 12

2.3 CONCEPT ANALYSIS 17

2.4 CONCLUSION 18

CHAPTER THREE - TECHNIQUES 19

3.1 SOFTWARE CLUSTERING TECHNIQUES 19

3.2 CONCEPT ANALYSIS TECHNIQUES 44

3.3 CONCLUSION 49

CHAPTER FOUR – SOFTWARE CLUSTERING ASSESSMENT 50

4.1 GENERAL SURVEYS 50

4.2 TZERPOS & HOLT 52

4.3 BAUHAUS 59

4.4 CONCLUSION 65

CHAPTER FIVE - ANALYSIS 66

5.1 METHODS OF ANALYSIS 66

5.2 CHOSEN TECHNIQUES 67

5.3 ANALYSED SOFTWARE 69

5.4 MOJO ADAPTATIONS 71

5.5 TOOL SUPPORT 73

5.6 CONCLUSION 76

CHAPTER SIX - RESULTS 77

6.1 COVERAGE 77

6.2 STABILITY 79

6.3 CASE STUDY 84

6.4 CONCLUSION 89

CHAPTER SEVEN - CONCLUSIONS **90**
7.1 HISTORY 90
7.2 WORK PERFORMED 91
7.3 RESULTS ANALYSIS 92
7.4 REVIEW OF SUCCESS CRITERIA 96
7.5 FURTHER WORK 98
7.6 CONCLUSION 100

REFERENCES **101**

LIST OF FIGURES AND TABLES

FIGURE 3.1: CALCULATING A NEW SIMILARITY 21
FIGURE 3.2: DENDROGRAM EXAMPLE 22
TABLE 3.1: EXAMPLE BINARY RELATION FOR CONCEPT ANALYSIS 45
TABLE 3.2: EXAMPLE SET OF CONCEPTS 46
FIGURE 3.3: CONCEPT LATTICE EXAMPLE 46
TABLE 5.1: SOFTWARE SYSTEM DETAILS 70
FIGURE 5.1 USE OF BAUHAUS RIGI 74
FIGURE 5.2: TOOL SUPPORT FOR ANALYSIS 75
FIGURE 6.1: COVERAGE GRAPHS 78
TABLE 6.1: ENTITIES AND CONNECTIONS BELONGING TO SOFTWARE SYSTEM 80
TABLE 6.2: PERCENTAGE CHANGE OF ENTITIES AND CONNECTIONS 80
TABLE 6.3: VERSION SIZES FOR MOJO CALCULATION 80
TABLE 6.4: ENTITIES OMITTED FOR MOJO CALCULATION 81
TABLE 6.5: MOJO RESULTS (MOJO(A,B) HIGHLIGHTED) 81
TABLE 6.6: MOJO RESULTS AS A PERCENTAGE OF VERSION SIZE 82
FIGURE 6.2: SYSTEM CHANGES AND MOJO VALUES AS PERCENTAGE OF SYSTEM SIZE 82
TABLE 6.7: PART TYPE CASE STUDY RESULTS 84
TABLE 6.8: DOMINANCE ANALYSIS CASE STUDY RESULTS 85
TABLE 6.9: SIMILARITY CLUSTERING CASE STUDY RESULTS 86
TABLE 6.10: CONCEPT ANALYSIS CASE STUDY RESULTS 88

ACKNOWLEDGMENTS

I would like to thank Dr. Rainer Koschke of the Bauhaus group for allowing me to use the Bauhaus Rigi toolkit and for answering my questions regarding the use of this toolkit. I would also like to thank the authors and owners of the software system used as a test system for this thesis, whose advice and information has been invaluable.

Above all, I would like to thank Dr. Elizabeth Burd, my supervisor, who has given me constant support and assistance throughout the writing of this thesis.

DECLARATION

Elements of this work have been published as follows:

Burd E., Bradley S., Davey J., 'Studying the Process of Software Change: an analysis of software evolution', Proceedings of the Seventh Working Conference on Reverse Engineering, pp 232-239, 2000

Davey J., Burd E., 'Evaluating the Suitability of Data Clustering for Software Remodularisation', Proceedings of the Seventh Working Conference on Reverse Engineering, pp 268-276, 2000

Davey J., Burd E., 'Clustering and Concept Analysis for Software Evolution', Proceedings of the International Workshop on Principles of Software Evolution 2001

STATEMENT OF COPYRIGHT

CHAPTER ONE – Introduction

As software, and the study of software development, becomes increasingly widespread, the problems involved in maintaining a software system are becoming more widely understood. A large amount of research is now focussed on the difficulties surrounding the evolution of software and on the development of techniques to support evolution. This thesis describes two types of method, software clustering and concept analysis, which could be used to assess the evolution of a software system.

This introductory chapter defines the problem area and describes the nature of the work detailed in the subsequent chapters. The criteria for the success of this work are listed and an overview of the rest of this document's structure is provided.

1.1 Problem

As the use of computers and computer software becomes increasingly widespread, it is becoming more and more important to understand the development of software. In particular, it is essential to understand how software that is in use over a long period of time changes as it is adapted to suit its users and surroundings. Any change to a system, whether during its original development or during its use, can affect the evolution of the system in ways that the system's developers and maintainers had never envisaged [TAKA96]. This evolution must be understood for the maintainers to plan and execute new changes to the system successfully.

For many years the term Software Maintenance has been used to describe the process of altering a system after it has been delivered to a customer. Software Maintenance is often under funded and rushed in execution, partly because there is still a perception that making such change is easy and the majority of the work is done during the original development of the system [TAKA96]. However, recently the term Software Evolution has been used to describe the development of software over the whole of its lifetime, from the project's inception through the original development until it ceases to be maintained [BENN96]. This forces the maintainers of the system to consider the history

and the future of the system rather than merely attempting to fix a problem in the short term, which could cause problems at a later date.

Therefore, there is a need for methods to study how a software system has evolved to assist the maintainers. The types of method presented here, known as software clustering and concept analysis methods, could demonstrate how a cohesive module of the system has disintegrated over time as changes have been made or how an individual function or data structure has become used for other reasons than were originally intended. In this way, when a change is required, the maintainers can make a more informed decision on how the system must be altered and on the best way to accomplish this.

Software Clustering is the application of Cluster Analysis to software systems. The aim of Cluster Analysis is to take an unsorted set of entities and cluster them together based on the features they have in common. For example, medical researchers use cluster analysis to group diseases based on the symptoms a sufferer exhibits [EVER93]. Software Clustering researchers have applied many existing cluster analysis techniques to software systems. The entities and features can be any number of quantifiable elements in a software system. Common examples of entities are functions, variables and data types and features can be any way of describing these entities; for example, functions may be described by their use of variables or their calls to other functions [WIGG97].

Concept Analysis techniques are very similar to Software Clustering techniques and are also based on entities and features of a software system. They were also developed outside of the software community and have only recently been applied to software. Concept Analysis techniques create sets of concepts based on a set of entities; these concepts are more formally defined than the sets of clusters created by software clustering.

Software clustering and concept analysis techniques have two main purposes; either as a way of suggesting restructurings of software systems, which may lead to some reengineering or reuse proposals, or as an aid to program comprehension. This study is the first to investigate their use for studying software evolution.

1.2 Proposed Work

This thesis extends the use of software clustering and concept analysis techniques for program comprehension by examining different versions of the same software system and attempting to discover evolutionary trends over the lifetime of the system. This work is being carried out because the techniques may help maintainers to specify detrimental trends and apply preventative maintenance to reverse them or to assess the likely impact of a proposed change. A number of different techniques have been assessed for this purpose.

In order for these techniques to be used for evolution, it is important that they should be stable from version to version; that is, the changes between representations of two versions should reflect the amount of change between the two versions. This is necessary so that the representations can be compared while still incorporating the major changes between versions. A measure called MoJo [TZER99] has been used to assess stability; this measure and an algorithm to calculate it is described in Chapter Four. A case study has also been performed, examining how a single group of entities develops over the lifespan of a system and how the techniques report this development.

1.3 Criteria For Success

The following criteria will be used to assess the success of this thesis. They will be reviewed in the conclusions in Chapter Seven.

1. To summarise the history of software clustering and concept analysis techniques.
2. To assess the coverage and stability of a number of software clustering and concept analysis techniques.
3. To use industrial strength software during the analysis of these techniques.
4. To develop tools to support the analysis of these techniques.
5. To investigate the feasibility of studying evolution of systems using these techniques.
6. To provide recommendations for the most appropriate techniques to use for the purposes of evolution.

1.4 Document Structure

The rest of this thesis is structured as follows:

Chapter Two introduces the background to the work described in the rest of the thesis and includes full definitions of Software Evolution, Software Clustering and Concept Analysis.

Chapter Three explains the history of Software Clustering and Concept Analysis research, detailing many of the significant techniques that have been developed.

Chapter Four outlines general work that has attempted to classify and evaluate the techniques detailed in Chapter Three and explains how this general assessment work motivated the work presented in this thesis.

Chapter Five describes the analysis that has been carried out for this thesis. A list of the techniques that have been studied in detail for this thesis is provided with explanations as to why these techniques were chosen. The tools that were developed to support this analysis are also discussed.

Chapter Six contains the results collected during the work described in Chapter Five. These results are explained in detail and some preliminary analysis of the results is included.

Chapter Seven presents the conclusions that have been drawn from the results that have been collected. There is a discussion on the work achieved based on the criteria for success outlined in Section 1.3. Finally, some of the further work that could build on the conclusions of this thesis is described.

1.5 Conclusion

This opening chapter has provided an introduction to the problem area and has briefly described the work that is reported by this thesis. The criteria for the success of this work have been defined and the structure of the rest of the thesis has been outlined. The following chapter describes the relevant fields of interest in more detail.

CHAPTER TWO – Background and Definitions

This thesis describes a survey into the use of software clustering and concept analysis techniques for software evolution. This chapter outlines the background to the fields of software clustering, concept analysis and software evolution and provides basic definitions of these terms.

2.1 Software Evolution

During the latter decades of the twentieth century, computers became a central part of most people's lives. In addition to the computers in the workplace which were common from the 1980s, development of mobile technologies and the Internet has meant that many leisure activities now involves a computer or a microchip of some kind. As shown by the Year 2000 problem, many aspects of society are now dependent on the correct functioning of computers [JONE98]. This, by extension, means the software that runs these computers must be robust and adaptable to changes in the outside world. Therefore, it is increasingly important that the evolution of computer software is understood and manageable.

The need for an understanding of software evolution has been reflected by a change of perspective by both industry and academia. There has been a proliferation of new conferences and journals, such as the International Workshop on Program Comprehension (IEEE), the Working Conference on Reverse Engineering (Reengineering Forum & IEEE) and the Journal of Software Maintenance: Research and Practice (Wiley), highlighting the need for research into the development of existing, rather than new, projects. van Deursen estimates that, since 1990, there have been more programmers working on enhancements and repairs than on new projects [DEUR99a].

There are four basic types of necessary maintenance for a software system, which many researchers have focussed on when investigating the continuing development of software systems. These types are as follows [LIEN80]:

Corrective maintenance: the fixing of defects present in the system
Adaptive maintenance: the adaptation to changes in the system's environment
Perfective maintenance: the introduction of enhancements to the system
Preventative maintenance: an effort to avoid malfunctions or improve maintenance

Despite the continuing importance of these four types of maintenance, an important development during the 1990s was a shift of emphasis from studying software maintenance as a separate phase of the software lifecycle to studying software evolution, which covers the entire software development process [BENN96]. It is now recognised that maintaining software is generally not a case of quick, simple fixes but can require significant design and comprehension work for changes to be implemented successfully, because software is continually changing and becoming more complex [LEHM97]. This substantial work is seen as more than simply maintenance, and is instead termed as *software evolution*.

The practices needed to control the evolution of software are still in their infancy, largely because the nature of this evolution is not well understood. Studies of evolution (such as work by Burd [BURD00]) have taken place but there is still much work to be done. A possible method of assessing the evolution of a software system is by using software clustering.

2.2 Cluster Analysis and Software Clustering

This section describes the multi-discipline field of Cluster Analysis and how the field of Software Clustering has emerged from this as a field of its own. A definition of Software Clustering and the background to this definition is also provided.

2.2.1 Cluster Analysis

Cluster analysis is a general term used to describe techniques for uncovering structure within complex sets of data. As a research area it is much older than Software Engineering; Sneath and Sokal [SNEA73] list papers from the turn of the twentieth century, and the field has developed greatly since (Anderberg [ANDE73] claims at least 600 relevant papers were published between 1960 and 1973). This wealth of information is due to the fact that, although traditionally cluster analysis methods have

been used for classification of species or medicines, most scientific disciplines and many social sciences have used similar clustering techniques. The burst of interest during the 1960s can be attributed to the computer, which allowed the implementation of previously theoretical algorithms.

Because cluster analysis does not belong to one particular science and there was no central repository for cluster analysis resources, researchers from each discipline tended to develop their own clustering techniques rather than learn from the techniques already in use. To counter this, since the 1960s, several general textbooks on the subject have been produced.

Sneath and Sokal produced the first summary of clustering techniques in 1963, entitled 'Principles of Numerical Taxonomy', which was updated in 1973 [SNEA73]. The book includes useful information on the history of numerical taxonomy, as well as a large selection of early references, mostly from biology. Most of the techniques used for cluster analysis today are presented, but many of the terms used in the book are unfamiliar and most of the terms used today were yet to be developed when the book was written, which means the descriptions are often difficult to read and understand.

Despite the problems with their descriptions of clustering techniques, still of interest today is Sneath and Sokal's identification of some of the advantages of clustering techniques. These include:

- the power to integrate data from a variety of sources,
- the automation of many parts of the classification process, and
- the objectivity of many techniques, disregarding preconceptions users may have about the dataset.

For Software Engineering, this means that very detailed descriptions can be used to quickly develop classifications that may give the maintainers a fresh perspective on their current mental model of the system.

Anderberg [ANDE73] presented a similar study of the field at the same time as Sneath and Sokal's book was published. This study is presented from a general perspective rather than the biological approach of 'Numerical Taxonomy', and, although it is not as comprehensive as Sneath and Sokal's book, the descriptions of techniques and development of ideas are well written and preferable for a novice user. Also, it was the

first study to approach the issue of which clustering techniques to use for a particular set of data, rather than simply describe the techniques in isolation.

Kaufman and Rousseeuw [KAUF90], like Anderberg, take a general view, preferring to concentrate on the types of clustering methods available rather than describe a large number of similar methods in detail. The text greatly benefits from the solidification of the clustering field since the publication of earlier textbooks and the descriptions are consistent and thorough. Each type of method is demonstrated by describing a computer program that implements the method, providing a good basis for the practical use of the techniques described.

The most recent general textbook is Everitt's 'Cluster Analysis' [EVER93], first published in 1974 but revised in 1980 and 1993. This book, Kaufman and Rousseeuw's book, takes a practical approach, describing not only the various clustering techniques but also guidelines on how to apply them. Unlike Kaufman and Rousseeuw, many different clustering methods are described but unfortunately the descriptions of individual techniques are often fairly short. However, the book still provides a valuable, current introduction to cluster analysis and provides enough references to allow the user to follow up on any useful techniques.

2.2.2 Software Clustering

Although cluster analysis has a long history, it was only recently that Software Engineering researchers became fully aware of this general cluster analysis work and began to apply existing clustering methods to software systems consciously. However, researchers have been developing their own clustering methods for many years, usually in an attempt to discover the structure of software systems. These techniques can be broadly termed *software clustering* techniques. The area of software clustering research will now be examined and the term software clustering defined.

Software clustering is based on structured programming methods. In one of the earliest papers on software structure, Parnas [PARN72] discusses the need for software to be modularised and suggests some criteria for the division of software into modules. He claims that modularised software, where functions are grouped together into small, cohesive units, is easier to understand, develop and maintain than unstructured software. This is because, if these modules are designed using appropriate criteria, changes can be

made to the design or the implementation of a single module without affecting the rest of the code.

Parnas defines the 'information hiding' criterion for module decomposition, where 'every module ... is characterised by its knowledge of a design decision which it hides from all others'. This idea has been developed and refined by many researchers, including Yourdon and Constantine, who identified two important properties of a module, cohesion and coupling [YOUR79]. Cohesion is the level to which the elements of a module are tightly bound together. Coupling is the degree of interdependence between modules. Ideally a module will exhibit low coupling and high cohesion. Yourdon and Constantine define the highest level of cohesion as functional cohesion, where every element of processing is an integral part of, and is essential to, the performance of a single function.

Most modern programming languages and design methods support these concepts and so many new software projects, if well designed, will exhibit the properties of high cohesion and low coupling. However, there is a large amount of software, including software written in modern languages, which does not exhibit these characteristics because of poor design or frequent, unstructured maintenance. This software, and indeed any 'critical software that cannot be modified efficiently' [GOLD98] is known as *legacy software*.

Working with and improving legacy software has become an important part of the software lifecycle and can be broadly defined by the term *reverse engineering*. Reverse engineering, as defined by Chikofsky and Cross in their landmark taxonomy of the area, is the process of analysing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction [CHIK90]

Software clustering methods are reverse engineering methods; however, they are a little more specific than the definition above suggests. In particular, they are usually concerned with clustering the system's components together and presenting these clusterings as a representation of the system.

The clustering process is not architecture recovery, which, as defined by Mendonça and Kramer [MEND97], is concerned with higher-level abstractions than most of the techniques presented here provide. Object recovery is a better description but is also unsuitable, as it leads to confusion with the term object in object-oriented programming and the image recognition area object recovery.

Koschke [KOSC00a] uses the term component recovery to describe the clustering process but, as with object recovery, this suggests a simple representation of the basic components of the system and does not naturally imply the grouping of these components. Although Koschke does expand the definition of component to include a cluster of components, it is felt this is not clear enough from a definition alone to use the term; indeed, Koschke never formally defines component recovery although he has used the term through a number of papers and his Ph.D. thesis.

Tzerpos [TZER98] introduces the term Software Botryology to describe the clustering of software systems (from the Greek word *botrys*, meaning a bunch (or cluster) of grapes). However, this term is not used again within the paper that introduces it, Tzerpos preferring to use the more common *software clustering*. This is the term that will be used throughout this thesis.

Software clustering is defined in this thesis as the identification of a software system's entities and their interrelationships and the grouping of these entities by some relevant criteria. There is a wide variety of software clustering methods, but there are basic elements common to all methods, established in a number of ways by Lakhotia [LAKH97], Wiggerts [WIGG97] and Koschke and Eisenbarth [KOSC00b], among others. The three elements are now described:

1. The *entities* that are to be clustered must be identified. An entity can be any element of the software system, including functions, variables, types, classes or any other identifiable structure.

2. The clustering *criterion* must be defined, to establish the nature of the resulting clusters. This usually includes which *features* should be used to describe the entities (for example, a function might be described by the global variables it

uses or the other functions it calls) and a definition of *similarity* between entities.

3. There will be a clustering *algorithm* that performs the clustering and usually determines the format of the resulting representation of the system.

Software clustering techniques aim to present a complete and consistent representation of a software system, usually as an aid to some form of software restructuring or program comprehension. However, it is often very difficult to cluster some entities in a system, either because they are very heavily used and could be placed in any number of clusters or because they are hardly used at all and do not belong in any existing clusters. There are various ways of dealing with these problems, which are discussed in the next chapter, but software clustering techniques will generally make decisions about where to position entities which provide a more complete representation of a system but may hide some information about the use of the entities in question.

A very similar field to software clustering is *concept analysis*. Strictly speaking, the field of concept analysis forms part of the field of software clustering according to the definition given above. However, the methods and results of concept analysis are sufficiently different to software clustering methods and results to warrant the discussion of concept analysis as a research area in its own right.

2.3 Concept Analysis

A field that has much in common with software clustering is *Concept Analysis*. Concept analysis attempts to define the complete set of logical groups (or *concepts*) in a set of entities. Methods for defining the concepts in a data set have been investigated for over sixty years but, as with cluster analysis, it is only recently that these methods have been used to uncover structure in legacy systems. The basic difference between software clustering and concept analysis is that software clustering is focussed on providing a representation of a whole software system whereas concept analysis aims to create the whole set of possible concepts within that system, regardless of the representation this concept set may provide.

A concept is a collection of entities and features such that the features in the concept represent the complete set of features that are used to describe the entities in the concept. The concepts defined by concept analysis may overlap with each other, which makes concept analysis unsuitable for restructuring work unless a partition of entities and features can be created from the set of concepts. Researchers have provided ways to create partitions from concept sets but these have generally been computationally very expensive and have frequently provided a large set of partitions with no way of choosing a suitable partition from the set.

Advocates of concept analysis claim that concepts can be used to overcome many of the problems encountered using cluster analysis. However, research has shown that there are also significant problems in the use of concept analysis to recover legacy system structure. Snelting [SNEL96] and van Deursen and Kuipers [DEUR99b] provide useful overviews of concept analysis; the latter paper is particularly interesting because it compares concept analysis to software clustering.

However, sets of concepts can be useful for program comprehension because no information is hidden from the user as it can be when using software clustering techniques. This may mean it takes more effort to understand a set of concepts than a set of clusters but it also means that decisions made on the basis of concept sets will often be better informed than decisions that are based on sets of clusters.

2.4 Conclusion

This chapter has explained the background to the fields of Software Evolution, Software Clustering and Concept Analysis and provided the basic definitions of these terms that will be used throughout this thesis. The following chapter describes a number of software clustering and concept analysis techniques in detail.

CHAPTER THREE - Techniques

This chapter outlines the history of software clustering and concept analysis by describing in detail many of the relevant techniques that have been developed since the early 1980s. The advantages and disadvantages of each technique are also explained. The techniques are discussed in the context of the researchers who proposed them to demonstrate how the field has developed since the basic need for such techniques was identified.

3.1 Software Clustering Techniques

3.1.1 Belady and Evangelisti

Belady and Evangelisti [BELA82] were among the first researchers to recognise the need for software clustering techniques for program comprehension. They propose a basic method for software clustering and a metric to quantify the complexity of a set of clusters.

The entities in Belady and Evangelisti's clusters are modules (functions) and control blocks (data structures). The system is described using a graph, the nodes of which are functions and data structures and each edge of which represents a connection between a functions and a data structure. Belady and Evangelisti do not describe these connections any further and so the precise nature of the connections is unknown.

Each function is given a unique number, as is each data structure. The entities and their relationships are then plotted on a separate graph, with data structures on the X-axis and functions on the Y-axis. A point is plotted for every edge between a function and data structure. This graph is then partitioned into clusters by attempting to place nodes that are close to each other on the graph in the same cluster.

Two parameters are identified which alter the resulting clusters. A limit is set on the maximum number of clusters produced and the maximum number of nodes in a cluster. These parameters are established to ensure the clusters that are produced are of a manageable size both for comprehension and maintenance.

In order to assist the determination of the values of these two parameters, Belady and Evangelisti describe a measure of complexity for clusters. This measure is based on the connections between nodes, subdividing the set of all connections into intercluster connections (edges which connect clusters together) and intracluster connections (edges which connect nodes within a cluster):

$$\text{Complexity} = \frac{1}{K}\frac{E_i}{E} + \frac{E_0}{E}$$

Here, K is the number of clusters in the system, E is the number of connections in the system, E_0 is the number of intercluster connections in the system and E_i is the number of intracluster connections.

While Belady and Evangelisti's [BELA82] clustering method and complexity measure have not been empirically tested, their work has been influential. The major flaw with the work is in the lack of definition of what a connection between two entities constitutes. However, the ideas they present, particularly with regard to complexity and intercluster/intracluster connections, have been frequently returned to (for example, the work of Anquetil and Lethbridge [ANQU99] or Canfora, Cimitile and Munro [CANF96]).

3.1.2 Hutchens and Basili

Hutchens and Basili [HUTC85] built on Belady and Evangelisti's work [BELA82] by developing a method based on data items shared by functions. They also introduced the use of hierarchical clustering algorithms, which have been frequently used in later work.

Their technique used *data bindings* to describe and cluster related functions. A data binding is relationship between two functions based upon a variable, defined as an ordered triple *(p,x,q)* where p and q are functions and x is a variable. There are four levels of data binding: potential, used, actual and control flow.

A potential data binding *(p,x,q)* exists if x is within the static scope of p and q. This means that there is a possibility of a data interaction between the two functions via x, although this interaction is not necessary for the potential data binding to exist.

A used data binding *(p,x,q)* is a potential data binding where *p* and *q* use *x* for either reference or assignment. This does not imply any closer connection between *p* and *q*, only that they use the variable *x* in some way.

An actual data binding *(p,x,q)* is a used data binding where *p* assigns a value to *x* and *q* references *x*. There is no consideration of the ordering of these events; the binding exists as long as both assignment and reference are executed at some point.

Finally, a control flow data binding *(p,x,q)* is an actual data binding where there is a possibility that *q* will be passed control after *p* has had control. This binding may exist even if *q* can never execute after *p* because of the dynamic properties of the program.

Once the relationships between entities are established, an algorithm must be used to cluster these entities together. Hutchens and Basili use an agglomerative hierarchical algorithm, which takes a bottom-up approach to clustering [HUTC85, WIGG97]. The clustering process begins with a set of clusters, one for each entity in the system, where each pair of clusters has a similarity value recording how similar the entities in the clusters are. The two most similar entities are clustered together and the similarity values between the new cluster and all other clusters are calculated based on the previous similarity values of the clustered entities. This process repeats until only one cluster remains, containing all the entities in the system.

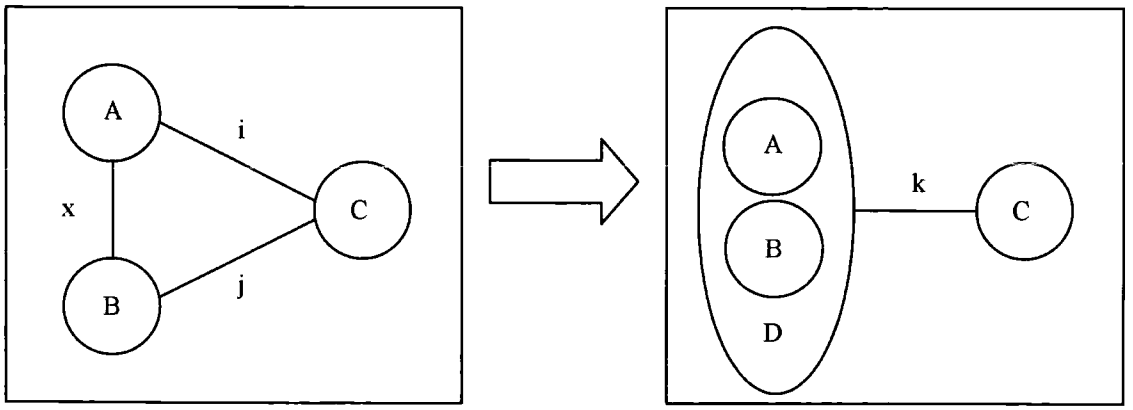


Figure 3.1: Calculating a new similarity

Agglomerative hierarchical clustering algorithms differ in the update rule they use to calculate similarities between new clusters and existing clusters. In Figure 3.1, the clusters *A*, *B* and *C* are linked by similarity values *i*, *j* and *x*. In this example, *x* is the highest similarity value and clusters *A* and *B* have been clustered together to form

cluster D. A new similarity, k, which determines how similar clusters C and D are, must now be calculated based on the values of i and j.

Two common update rules are single linkage and complete linkage. Single linkage means the new similarity k becomes the largest value of i and j. Complete linkage means k becomes the smallest value of i and j. Hutchens and Basili use a weighted average update rule to determine this new similarity. This takes into account the number of entities in the two clusters that are being joined together, giving more weight to the similarity value that represents the most entities. For example, in Figure 3.1, if A contained ten entities and B contained five, j would be given twice the weight of i when the new similarity was calculated.

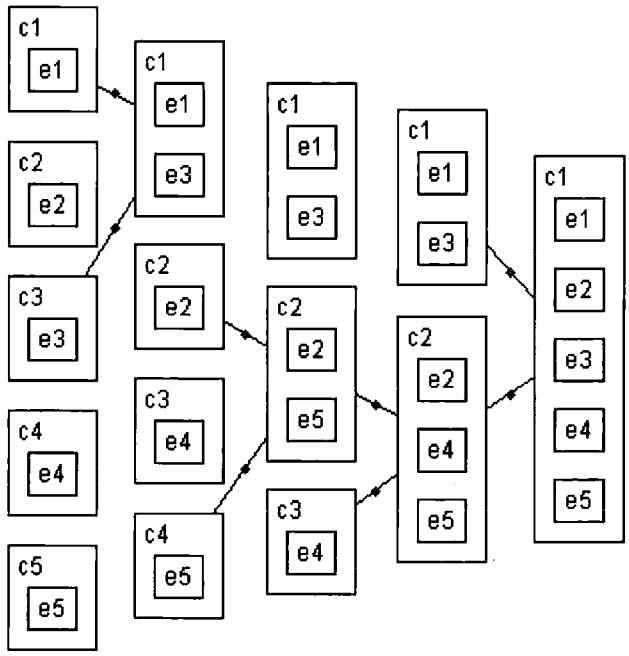


Figure 3.2: Dendrogram example

The results of hierarchical algorithms are traditionally presented as a dendrogram. This is a pictorial representation of the clustering process, showing each clustering which was created by the algorithm, from the initial 'one entity, one cluster' clustering to the final 'all entities in one cluster' clustering. Figure 3.2 shows a dendrogram where five entities (*e1* to *e5*) are placed in individual clusters (*c1* to *c5*). They have been clustered together, joining two clusters at a time, until they all belong in the same cluster.

The example shown in Figure 3.2 is a slight adaptation of a traditional dendrogram. Usually, only lines will be drawn to denote the joining of two clusters and the actual

clusters will not be printed. However, this makes it difficult to examine a particular clustering without backtracking through the dendrogram. In Figure 3.2, each column represents a single clustering, which makes it easy to isolate and examine each one.

While the dendrogram is a good record of the process, and provides some information on how strongly entities are clustered together, further work is needed if a single representation of the system structure is required (for example, if the system was to be restructured). With no concrete way to compare one clustering to another, the selection of a single, representative clustering from the set of clusterings presented is still a major problem in software clustering [WIGG97].

Hierarchical algorithms are very popular with software clustering researchers. Hutchens and Basili [HUTC85] chose a hierarchic method because they believed that 'systems and programs are best viewed as a hierarchy of modules'. This may account for their intuitive appeal, but the results gained are often poor, especially considering the difficulties regarding the partitioning of dendrograms.

Part of the problem with Hutchens and Basili's approach is their choice of a dissimilarity metric as their clustering criterion. Dissimilarity (or distance) metrics work by considering the absence of a descriptive feature to be a mark of similarity. These metrics are often used in other disciplines, such as classification of species, where the absence of a particular defining feature may well be as significant as the presence of a feature in a pair of entities. However, for software clustering, such metrics are inappropriate because they do not reflect the nature of software. For example, the fact that two functions do not use a certain data item does not suggest any similarity between the functions. Dissimilarity metrics have been used by a number of software clustering approaches but are now generally considered unsuitable [WIGG97, DAVE00].

3.1.3 Choi and Scacchi

Choi and Scacchi [CHOI90] define a clustering method based on graph theory that uses files as entities, reasoning that programmers usually group related functions into a file. This logic is particularly valid for very large software systems when it is often not computationally feasible to use functions as entities. The entities are described using a resource flow diagram (RFD), which is an undirected graph. The nodes in an RFD are the entities (files) and there is an edge between any pair of entities that share resources.

For example, if one entity calls a function or uses a data item belonging to another entity, there is an edge between the two entities. In the RFD, however, no record is made of the resource shared or the direction of the sharing – only the fact that two entities share resources is recorded.

This RFD is used to create a resource-structure diagram, or RSD, which is a tree containing entities and clusters, where entities form the leaves of the tree and clusters form the inner nodes of the tree. The children of a cluster node are entities or other clusters that belong to the cluster node. This hierarchical structure provides a view of the system's architectural design.

Choi and Scacchi [CHOI90] provide an algorithm for converting an RFD to an RSD, which is based on two key properties: minimum coupling and minimum alteration distance. Minimum coupling is as defined by Yourdon and Constantine [YOUR79] in Section 2.2.2. If two entities X and Y are modules, and when X is altered Y is affected, the alteration distance between X and Y is the length of the path in the RSD that joins X to Y. If X and Y belong to the same subsystem the alteration distance between them is 0. In order to localise changes to the system the alteration distances between entities should be as low as possible.

The RSD is a reasonable high-level representation of a system and could provide the basis for a restructuring of a software system. However, the information used to partition a system and the way the partitioning is performed is very coarse. As with a dendrogram representation (which is very similar to an RSD) bad decisions early in the process can affect the form of the entire diagram. This is a particular problem because a great deal of information about how resources are shared and the importance of various resources, which could assist the avoidance of such bad decisions, is not included in the analysis. However, as a way of providing a preliminary high-level description of a system before some more thorough analysis is performed, Choi and Scacchi's method may prove useful.

3.1.4 Liu and Wilde

Liu and Wilde [LIU90] propose two different methods for software clustering, both based on data structures. Drawing from object-oriented programming, and the belief

that one of the greatest maintenance challenges is the understanding of system data, two ways of identifying candidate objects are presented. These objects have the form

Candidate Object = (F,T,D)

where F is a set of functions, T is a set of types and D is a set of data items. Any of these sets can be empty and ideally they will be disjoint from the sets in other candidate objects (although this is not required).

The first method proposes candidate objects based on global data items, whereby if two functions use the same global data item they will be clustered together. The second method uses types instead of global data items. Type x is defined as a sub-type of type y if x is used to define y; y is then a super-type of x. Functions are clustered together if they both use the same super-type, which may be defined by a use of a sub-type.

Three problems common to software clustering methods are highlighted by Liu and Wilde's work [LIU90]. Liu and Wilde themselves note that these methods often produce unsatisfactory 'objects', or clusters, and human intervention may be needed to adjust the clusters. Many researchers since have noted that automatic clustering methods can be greatly enhanced by alteration of the results by maintainers [SCHW91, MANC99, KOSC00a].

Liu and Wilde's work also demonstrates the possibility that overlapping clusters will be produced. If a software system's functions are badly encapsulated (as will usually be the case for legacy software, especially the legacy software that clustering methods will be used for) there will often be situations where one function could be part of more than one cluster or even not fit neatly into any cluster. This can either be dealt with by allowing overlapping clusters, where one function can be part of two or more clusters, or by allowing only disjoint clusters and forcing functions into one cluster only. The former can produce better results but requires intervention from the maintainers of the system to specify where functions in more than one cluster should actually be. The latter is automatic, but can often require a maintainer to reposition functions that have been wrongly clustered. Most approaches use disjoint clusters, but some have attempted to deal with the problems with this approach (such as Schwanke's Arch project [SCHW91], described in Section 3.1.5).

Finally, Liu and Wilde's approaches are typical of many software clustering methods in that they focus on a single aspect of a software system rather than incorporating a wider picture of the system to perform the clustering. This means that the clusters are often unrepresentative of much of the system as they ignore critical information, even if the motivation for the clustering is sound (for example, to expose the data structures of a system). Furthermore, such limited approaches are unsuitable for some software if the aspect they exploit is not used in depth by the software. Liu and Wilde point out that well written software will not contain many global variables, thus making their first approach useless for such software. Incorporating more information is difficult because the computational load increases with every extra aspect of the system examined, but recent approaches have tended to use multiple system views because single aspect views have not usually produced universally good results [ANQU99].

Liu and Wilde's work was later developed by Livadas and Johnson (see Section 3.1.7).

3.1.5 Arch

Schwanke's Arch project [SCHW91], developed in the early 1990s, has been hugely influential for subsequent software clustering researchers. It developed and formalised the generic structure of software clustering methods suggested by earlier work and proposed a number of innovative developments to build on this structure. Significantly, Schwanke highlights the need for human intervention into clustering processes, because the domain of a software system has a huge effect on the success of clustering methods and many methods produce imperfect results that benefit greatly from adjustment by users of the system.

Schwanke bases his work on Parnas's information hiding principle [PARN72] (discussed in Section 2.2.2) and defines an information sharing heuristic for detecting when two procedures share a design decision:

- If two procedures use several of the same unit-names, they are likely to be sharing significant design information, and are good candidates for placing in the same module.

The following similarity metric was developed based on this heuristic (refined by Girard and Koschke [GIRA99]):

$$Sim\ (A,B) = \frac{Common\ (A,B) + k * Linked\ (A,B)}{n + Common\ (A,B) + d * Distinct\ (A,B)}$$

Common(A,B) is the size of the set of common features of entities *A* and *B*. *Distinct(A,B)* is the size of the set of distinct features, which is the set of features used by *A* but not *B* and the set of features used by *B* but not *A*. *Linked (A,B)* is 1 if *A* calls *B* or *B* calls *A* and 0 otherwise. *d* and *k* are constants which control the relative importance of *Linked(A,B)* and *Distinct(A,B)*.

The constant *n* controls normalisation. If *n* is 0, all similarities are normalised between 0 and 1. This is common to many software clustering techniques because it makes it easier to understand and compare similarity values. However, if *n* is greater than 0, similarities can be any value. This means that a pair of entities that agree on a large number of features will be classed as more similar than a pair of entities that agree on a small number of features, whereas these pairs may have been given the same similarity value if these values had been normalised.

Arch was also the first project to use weighted features. Schwanke's hypothesis (based on earlier clustering work in other disciplines) was that agreement on rare features is more important than agreement on common features [SCHW91]. This is particularly interesting for software clustering because the fact that some functions and data items are very infrequently used often suggests they perform some specific purpose that would be ideal for encapsulation. Therefore Arch estimates the significance of a feature by its Shannon information content:

$$Weight(f) = -log(Probability(f))$$

The probability of *f* is the number of procedures that have feature *f* divided by the total number of procedures.

Arch supports a basic agglomerative hierarchical clustering algorithm with a single linkage update rule and three variations on this algorithm, which go some way to aiding the dendrogram partitioning problem. These are batch clustering, where a standard dendrogram is produced and 'useless' groups are heuristically removed, interactive, radical clustering, where the Arch user is asked for confirmation after each clustering, and interactive clustering, which compares a previous classification to the current

clustering process. These developments help to prevent one of the common problems in hierarchical clustering, where incorrect early decisions can produce useless results. By examining these decisions as the process is executed, an attempt can be made to reject the errors.

Schwanke also introduced the idea of maverick analysis. A maverick is an entity that has been clustered incorrectly and maverick analysis attempts to relocate these entities. It does this by examining an entity's neighbours, which are other entities with which it has at least one common feature. If a majority of its k nearest neighbours are bad neighbours (neighbours which do not belong in the same module as the entity) the entity is a maverick and needs to be relocated to be with its good neighbours. Schwanke found that this analysis coupled with other examinations by system analysts greatly improved the clusterings produced by the automatic Arch algorithms, and later used the nearest neighbour approach as a clustering method [SCHW94].

3.1.5 Patel

Patel, Chu and Baxter [PATE92] developed a method of clustering functions according to the types they use. The method is actually defined as a way of measuring the cohesion of a cluster but it can also be used in an attempt to generate cohesive clusters. Similarity between entities is calculated by counting the number of times each non-local type in a system is used by the entity. The use of a type T can be:

- accessing or setting a variable of type T, whether the variable is global or local
- having a parameter of type T
- using variables or parameters belonging to a subtype of type T

This approach is not normalised, so entities that use the same type many times are considered more similar than entities that only use a type a small number of times.

For any entity X a vector $(x_1, x_2, \ldots x_i, \ldots x_{n-1}, x_n)$ is created, where x_i is the number of times X uses a type i. For any two entities X and Y, the similarity of X and Y is given by:

$$Sim\ (X,Y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \times \sqrt{\sum_{i=1}^{n} y_i^2}}$$

The cohesion of a cluster of entities $E=\{e_1, e_2, \ldots e_m\}$ is the average of the similarity measures over distinct pairs of entities in the cluster. This is calculated with the *Composite Module Cohesion* measure:

$$Cohesion\ (E) = \frac{\sum_{S} Sim(e_i, e_j)}{\sum_{i}^{m-1} i}$$

where $S = \{ (i,j) \mid i,j \in [1,m] \ \& \ i > j \}$. This calculation of S ensures that each pair of entities is only used once in the calculation, as opposed to using $Sim(e_i, e_j)$ and $Sim(e_j, e_i)$.

Patel et al's technique is essentially a specialised version of Schwanke's Arch method [SCHW91], in that it examines common features of entities. However, the examination of types as features of entities is very limited (as Liu and Wilde's work [LIU90] was limited) by ignoring a large amount of information about the entities. Although the idea that two functions that use the same types will have a similar purpose can be a valid one, this has not been empirically validated. Schwanke's approach was also subtler in it's weighting of features.

3.1.6 Rigi

One of the most commonly used reverse engineering tools is Rigi, developed by Müller and others [MÜLL93]. The aim of Rigi is to provide comprehensive techniques for discovering, restructuring and analysing software structures. This involves not only forming clusters from the entities in the system but also specifying the interfaces between the clusters and creating different views of the system for certain target audiences.

The basic representation form used by Rigi is a Resource Flow Graph (RFG), which is the same as the Resource Flow Diagram used by Choi and Scacchi [CHOI90] (see Section 3.1.3). There is no restriction on what the entities in the graph may be, as there was for an RFD. Two sets are identified based on this graph for each entity E: $Prv(E)$, which is the set of all syntactic objects provided by E, and $Req(E)$, which is the set of all syntactic objects required by E.

Based on these sets a Composition Dependency Graph (CDG) is created, where nodes are subsystems (which can be functions, classes, data types or groups of these entities)

and edges are aggregation or composition relationships between the subsystems. Entities are not restricted to a single subsystem, which means that multiple hierarchies can be represented using one model. The interfaces between two subsystems X and Y is a pair of sets *ER(X,Y)* and *EP(X,Y)*. *ER(X,Y)* is the set of exact requisitions of X from Y, which is the union of *Req(X)* and *Prv(Y)*. *EP(X,Y)* is the set of exact provisions of X to Y, which is the union of *Prv(X)* and *Req(Y)*.

Once these subsystems and subsystem interfaces are in place, their quality is assessed using a number of measures. Firstly, the interconnection strength *IS(X,Y)* assesses how strongly coupled two subsystems X and Y are. This is defined as:

$$IS(X,Y) = |ER(X,Y)| + |EP(X,Y)|$$

Thresholds can be established and altered by the user of Rigi to establish what level of coupling is acceptable. Rigi also uses a measure to determine the common providers (clients) and required subsystems (suppliers) of a set of subsystems, in the hope that some of this set could be merged to reduce the interfaces between subsystems. For a set of subsystems M in an RFG *G=(E,C)*, the common client subset *CS(M)* and the common supplier subset *SS(M)* are defined as follows:

$$CS(M) = \bigcap_{x \in M} \{e \in E \mid < x, e > \in E\}$$

$$SS(M) = \bigcap_{x \in M} \{e \in E \mid < e, x > \in E\}$$

Rigi has become popular among software clustering researchers because it allows the user to incorporate their own clustering algorithms into the tool, while providing excellent visualisation methods and the basic metrics described above [MÜLL93, KOSC00a]. However, it is unlikely that the average software maintainer will have the desire or time to adapt and use a tool in this way.

3.1.7 Livadas and Johnson

Livadas and Johnson [LIVA94] developed the earlier work of Liu and Wilde [LIU90] (see Section 3.1.4). They construct a framework for clustering, whereby a set of primary clusters is created automatically and a set of secondary clusters is created by the user refining the primary clusters. Three types of primary cluster identification methods are described: *global-based*, *type-based* and *receiver-based*. The first two approaches are

developments of Liu and Wilde's methods; Livadas and Johnson designed the receiver-based approach.

The global-based approach, as described by Liu and Wilde, clustered functions that used the same global variables. Livadas and Johnson describe two types of variable use that can be considered global but are not included by Liu and Wilde's method. Firstly, if a language permits nested procedures (as Pascal does, for example) any variables defined by a procedure P are considered global by any procedures nested within P. Secondly, if a global variable is passed as a parameter to a function, that function can be considered to use the variable. Livadas and Johnson also define the *global-static* approach for C code, whereby static variables are considered global and included in the analysis.

Livadas and Johnson demarcate the type-based approach into three separate approaches; clustering based on the types of a function's parameters (*parameter-type)*, clustering based on the types of a function's return value (*return-value-type*) and clustering based on the types of global and static variables used by the function (*global-type*).

The receiver-based approach was provoked by the difficulty in assigning a function to a cluster when the variables and parameters it uses are spread over a number of clusters. Therefore, a *receiver-parameter-type* is defined as the type of a parameter of a function F that is modified in at least one execution path of F. By clustering according to receiver types, the chance of clustering functions with the types that they are most concerned with is increased. A *receiver-global-type* is also defined, allowing global variables (and static variables in C) to be used as receivers.

Once these primary cluster identification methods have been run, the secondary methods can be used to collate and refine the results. Livadas and Johnson define a number of secondary methods, mostly based on relational database queries. For example, the *union* method combines the results of primary methods; the *selection* method only runs on a subset of functions and the *deletion* method allows the user to delete dependencies that do not reflect the true nature of the system. Liu and Wilde's supertype structure is also used to refine primary clusters that contain many types; by removing subtypes and leaving only the supertype, the cluster can be more clearly defined.

These refinements provide a more comprehensive approach to software clustering, especially by allowing the user to combine different views of a system. However, once again, the authors did not empirically test the approach and no real justification is given for the use of the methods they define. Furthermore, constructs such as pointers are not included by the approach. Although further work on Livadas and Johnson's representation methods, which include pointers, has been published, this has never been extended to the clustering methods they developed.

3.1.8 Yeh

Yeh, Harris and Rubenstein [YEH95] created an analysis tool, OBAD, which attempted to uncover abstract data types (ADTs) and objects in a software system written in a procedural language. An abstract data type is defined as 'one or more related data representations whose internal structure (private area) is hidden to all but a small group of procedures, i.e., the procedures that implement that abstract data type'. An object is defined as 'an entity which has some persistent state (only directly accessible to that entity) and a behaviour that is governed by that state and by the messages the object receives'. This means that objects are instances of abstract data types.

Abstract data types are based around record types of a system and their internal fields and objects are based around global and other external variables. OBAD recovers ADTs and objects by building abstract syntax trees (ASTs) to represent the system. For ADTs, the nodes of the tree are record types and functions that use these record types and the edges are references by the functions to internal fields of record types. For objects, the nodes of the tree are functions and external variables and the edges are references by the functions to external variables. These approaches can be combined to create entities made up of record types, global variables and functions.

Yeh et al [YEH95] list a number of problems with this approach, notably that OBAD often extracts ADTs and objects from library code which is not part of the main software system and also that the entities created when using the combined approach are often very large. Unfortunately, no solutions are offered to these problems other than to recommend that the users of OBAD examine and alter the resulting ADTs and objects themselves.

Also, OBAD does not attempt to partition a system; it only extracts as many ADTs and objects as it can find, which means the users of OBAD will have to collate this information to gain a full picture of the system. This could potentially take a long time, especially since legacy systems will have very convoluted and probably very large ADTs and objects.

OBAD doesn't take into account non-record entities (such as pointers, arrays and strings) either, although Girard and Koschke [GIRA00] describe ways to incorporate internal access to these entities. For example, a function which uses any index subscript of an array or any dereference of a pointer could be grouped with that array or pointer.

3.1.9 Canfora

Canfora, Cimitile and Munro [CANF96] developed a software clustering technique that uses a new metric, similar to Belady and Evangelisti's complexity measure, to assess the quality of a set of clusters. Their method is unique because, rather than attempting to simply reconfigure the clusters the method creates, the algorithm used actually alters the functions in the system to create more cohesive clusters.

The system is represented using a *variable-reference graph*. A variable-reference graph is a directed bipartite graph where the nodes are functions and global variables and the edges represent uses of global variables by functions; these edges are directed from function to global variable. Canfora et al propose that, ideally, this graph should have the form of a set of isolated sub-graphs, each of which consists of a single global variable node and one or more function nodes that use the global variable.

However, this is very rarely the case, as noted by Livadas and Johnson [LIVA94], because functions often use a wide range of global variables, especially in heavily maintained legacy systems. In particular, Canfora et al establish two types of undesirable links between functions and global variables: coincidental connections, which are the result of a function having two distinct purposes that use different global variables, and spurious connections, which occur when a function has a single purpose but the data structures supporting that function are badly designed.

Canfora et al [CANF96] propose an algorithm that refines a system's initial variable-reference graph by breaking bad connections until the graph has the form described

above. This algorithm uses Canfora et al's Internal Connectivity (IC) metric, which is defined as follows.

For each node n in the graph, where N is the set of global variables and functions that make up the nodes of the graph and E is the set of edges that join them, two sets are defined:

$$preSet(n) = \{y \mid y \in N \wedge (y,n) \in E\}$$
$$postSet(n) = \{y \mid y \in N \wedge (n,y) \in E\}$$

If n is a function, *preSet(n)* will be empty and *postSet(n)* will consist only of global variables. If n is a global variable, *preSet(n)* will consist only of functions and *postSet(n)* will be empty. Following the definition of these sets, a set of subgraphs is created, one for each function in the graph. A subgraph of a function f is the set of variables f uses and the set of functions that use those variables, described as follows:

$$V(f) = postSet(f)$$
$$F(f) = \bigcup_{d \in postSet(f)} \{f_i \mid f_i \in preSet(d) \wedge postSet(f_i) \subseteq postSet(f)\}$$

Based on these sets, the internal connectivity index of the subgraph of function f is defined as the ratio between external and internal edges of the subgraph of f:

$$IC(f) = \frac{\sum_{d \in postSet(f)} \#\{f_i \mid f_i \in preSet(d) \wedge postSet(f_i) \subseteq postSet(f)\}}{\sum_{d \in postSet(f)} \# preSet(d)}$$

This index is used to calculate the array ΔIC, which contains an entry for each function. ΔIC(f) is the difference between IC(f) and the IC value for all subgraphs merged together:

$$\Delta IC(f) = IC(f) - \sum_{d \in postSet(f)} \frac{\#\{f_i \mid postSet(f_i) = \{d\}\}}{\# preSet(d)}$$

The ΔIC value of a function is used to assess the cohesiveness of the function and the elements it is connected to. If the ΔIC value is high, the subgraph of the function is clustered into a single node. If it is low, the function is sliced into a number of separate functions, one for each global variable it uses. Each of the new functions will represent the old function's use of a single global variable. The threshold for ΔIC that determines whether a function should be clustered or sliced is specified by the user of the system;

Canfora et al suggest this is assisted by assessing the cohesiveness of a subset of the whole system before using the algorithm [CANF96].

As stated earlier, this calculation of the ΔIC array and the subsequent clustering of cohesive functions and slicing of incoherent functions continues until the variable-reference graph is a collection of isolated subgraphs containing one global variable and the functions that use it.

This approach was tested on very small software systems (<10KLOC) but shows promise based on these tests. It successfully identifies functions with low cohesion and attempts to adjust the system to improve this cohesion and avoids the bad decisions made during other techniques' clustering processes by continually assessing the state of the system graph. However, as with many other techniques, the process may need some adjustment by the user in order to produce good decompositions of the system. The major flaw with the method is that, although slicing functions to improve cohesion is possible for a reuse or reengineering approach, it is completely unacceptable for a program comprehension approach, where the system must remain intact. Further work on this method has been done to overcome this flaw [CANF00] (see Section 3.1.13.3).

3.1.10 Cimitile & Visaggio

Software clustering methods have been used for reuse reengineering, which attempts to create reuse components by extracting cohesive clusters from legacy code. One of these approaches is Cimitile and Visaggio's dominance tree analysis [CIMI95]. This method is based on functional abstractions of a system.

Cimitile and Visaggio use aggregation, which links several functions together if they have a combined purpose (as opposed to Canfora et al's isolation, where functions with multiple purposes were sliced into separate functions with a single purpose [CANF96]). This is done by creating a graph of the system, where nodes are functions and the directed edges are all the calls from one function to another, and converting this call graph to a dominance tree.

Cimitile and Visaggio define the dominance relation between functions as follows:

> If *px* and *py* are two nodes in a call directed graph, then *px* dominates *py* if *px* is in all paths in the graph from the start node *s* to *py*.

The start node *s* dominates all other nodes and so is the root of the dominance tree. By constructing a dominance tree, it is possible to see how the system's functionality is implemented by the functions in the system and how these functions can be clustered. Once the dominance tree has been created, the user can select subtrees of this tree to form clusters (or reuse candidates) of the system.

There are two different types of dominance relation: direct dominance and strong dominance (also called strong direct dominance). *px* directly dominates *py* if, for all nodes which dominate *py*, they also dominate *px*. This means every node in the call graph will have a unique direct dominator (apart from *s*) and, if *px* directly dominates *py*, there will be an edge (*px*,*py*) in the dominance tree.

px strongly dominates *py* if *px* directly dominates *py* and *px* is the only node in the call graph calling *py*. It can be inferred from this that if *px* only directly dominates *py*, other nodes call *py* but these nodes are not as dominant over *py* as *px* is.

The major drawback of this approach is that cycles cannot be represented in the dominance tree because each node in the cycle must dominate every other node. Therefore, these cycles are collapsed into a single cluster before the dominance tree is created. If there are large cycles in the system, which can commonly occur in legacy systems, there is a large loss of information to the clustering process.

In Cimitile and Visaggio's original description of the dominance relation [CIMI95], only the call structure of the system is examined and the data structures of the system are not considered in any way. This is out of step with most software clustering methods, which advocate an object oriented approach to clustering, and the results achieved using dominance trees in this way have not been particularly promising (Cimitile and Visaggio's experiments resulted in only 42% of the source code being covered by clusters found using the dominance tree). However, the approach has been extended to include data structures by Koschke [KOSC00a], which has produced more detailed results.

3.1.11 Bunch

Bunch ([MANC98], [MANC99], [DOVA99]) is a tool developed by Mancoridis and others that implements a graph-based software clustering technique. Bunch uses a

Module Dependency Graph (MDG) to represent the software system to be analysed. The nodes in an MDG can be any entity in the system including classes and functions but are usually files. An MDG has directed edges that represent the relationships between entities; these relationships can be anything that can be extracted from the source code [MANC98].

Using files as entities allows large systems to be analysed more efficiently than using functions as entities. Descriptions of files can be much richer than descriptions of individual functions because they form a larger, more complex code sample. However, it assumes that the functions and data items in a file are located correctly, as there is no way to change the contents of the files themselves, only the way the files are clustered together. This is often a necessary assumption because using functions as entities in a very large system is not computationally realistic.

Bunch attempts to partition the MDG meaningfully using a genetic algorithm, which is an alternative to more common hill-climbing algorithms. During the execution of a hill-climbing algorithm, it is possible to reach a local optimum, where there is no clear superior next step in the process. Genetic algorithms overcome this by randomly altering the input data to the algorithm insignificantly, which resolves the conflict between two identical clusters [DOVA99].

The metric used to run the genetic algorithm is based on inter-connectivity and intra-connectivity indexes, similar to those developed by Belady and Evangelisti [BELA82] (see Section 3.1.1) and Canfora [CANF96] (see Section 3.1.9). However, Bunch's metric is coarser than previous definitions because it does not count the number of dependencies between modules; this means that if module *A* has one link to module *B* and one hundred links to module *C*, *B* and *C* are considered equally similar to *A*. The authors claim they are redefining inter-connectivity and intra-connectivity to take account of these weightings [MANC99].

Bunch provides a useful approach to software clustering and the authors have achieved good results with their method [MANC99], although it has not been compared to more traditional hierarchical algorithms and so the value of using a genetic algorithm is unknown. Also, the use of Bunch may be restricted to a certain size of software system; small systems will usually not contain enough files for the process to be worthwhile and

the computational intensity of genetic algorithms could possibly make the method inadvisable for very large systems.

3.1.12 Tzerpos & Holt

Tzerpos and Holt's research has focussed on more general assessment of software clustering techniques rather than the proposal of their own techniques; this general work is discussed in more detail in Section 4.2. However, based on their general work, they have proposed a new software clustering technique known as ACDC [TZER00b].

Tzerpos and Holt have noted that many software clustering researchers are focussed on improving the performance and accuracy of software clustering techniques, sometimes at the expense of the comprehension of the representations these techniques produce [TZER00a]. The ACDC algorithm, therefore, is focussed on improving the comprehension of the results of such a method. Three features of a comprehension-driven clustering process are described:

- *Effective cluster naming*: clusters should be named according to their contents rather than by some arbitrary numeric scheme.
- *Bounded cluster cardinality*: very cohesive clusters are no good if they are very large and difficult to understand. Therefore, clusters should be limited to an easily comprehensible size.
- *Pattern-driven approach*: certain patterns often occur when humans create decompositions of legacy systems and so decompositions containing these patterns are easier to comprehend.

A list of possible patterns is provided, including a source file pattern, where all elements of a source file are placed in one cluster, and directory pattern, where all elements from a subtree of the directory structure are placed in one cluster.

The ACDC algorithm has two stages: skeleton construction and orphan adoption. Skeleton construction examines the occurrence of different patterns in the source code and creates clusters as the patterns are uncovered. The clusters are named based on the filenames of the files they originate from.

Orphan adoption [TZER97] attempts to place any entities that have not been clustered using skeleton construction and reassign any entities where there is slim justification for their current clustering. Once the set of orphans have been identified, an algorithm is used to incorporate orphans into the skeleton clustering of the system. The algorithm is based on a range of formal and informal criteria and each orphan is placed with whichever cluster depends upon it most according to these criteria.

Tzerpos and Holt achieved better results with ACDC than most other software clustering algorithms; this is particularly notable because, unlike many software clustering methods, ACDC has been tested on large software systems such as Linux. However, these results were achieved by comparing the clusterings ACDC produced with the expert clusterings of the maintainers. As ACDC is largely based on the programmer's interpretation of the software's structure (through file naming and directory construction) these better results are to be expected.

It is felt that, in terms of program comprehension, one of the main aims of software clustering techniques should be to discover the *actual* structure of the system from a low level in order to uncover flaws in the programmers' current mental model. While ACDC can be used as an input to this process, it is felt that it is not sufficient to rely on it as a software clustering method.

3.1.13 Bauhaus

The Bauhaus group, led by Koschke and Girard, have proposed a number of new software clustering techniques as part of their evaluation of software clustering, which is described in detail in Section 4.3. Koschke and Girard's software clustering techniques are based on their ability to extract three basic kinds of low-level components:

- *ADT – abstract data type*: an abstraction of a type which encapsulates all the type's valid operations and hides the details of the implementation of those operations by providing access to instances of such a type exclusively through a well-defined set of operations
- *ADO – abstract data object*: a group of global variables together with the functions that access them
- *HC – hybrid components*: cross-breedings of ADTs and ADOs containing functions, variables and types.

The main distinction between an ADT and an ADO is that there can be many instances of an ADT but only one instance of an ADO in a software system. Koschke and Girard consider these as the smallest components that are significant at the architectural level [GIRA00]. The term *related subprograms (RS)* is also defined as a set of subprograms that together perform a logical function and so have functional cohesion. The four new techniques defined by Koschke and Girard are now described.

3.1.13.1 Same Module Heuristic

The Same Module heuristic, an original approach by Koschke and Girard [GIRA00, KOSC00a], creates ADOs and ADTs by grouping functions with types or global variables that belong to the same module or file as the functions. This is similar to the basis of Tzerpos and Holt's ACDC algorithm [TZER00b] (see Section 3.1.12). The heuristic assumes that the programmers of the software system have generally used good information hiding principles throughout the evolution of the system, which is often not the case.

3.1.13.2 Same Expression Heuristic

Koschke [KOSC00a] also proposed the Same Expression Heuristic, which considers variables or parts of types that are used in the same expressions. Koschke proposes clustering entities with variables or types that they use if those variables or types appear in the same expressions. These expressions can either be statements within a function or parameters in the call to a function. Clusters that are produced using this approach that only contain one entity are disallowed.

3.1.13.3 Revisited Delta IC Approach

Canfora, Czeranski and Koschke [CANF00] extended the Delta IC approach developed by Canfora et al [CANF96] (see Section 3.1.9) to make it appropriate for program comprehension rather than restructuring by disallowing the slicing of functions with a low ΔIC value. The approach is also extended to cover types and cohesion.

To avoid the slicing of poorly constructed functions, the revised approach to Delta IC allows overlapping candidates to remain in the representation while the clustering process is being executed. These overlapping candidates are then merged depending on the level of overlap between them.

The original Delta IC approach extracts ADOs only; the revised approach is extended to extract ADTs as well. ADOs hide global variables that are only manipulated by functions in the ADO. ADTs hide the underlying data structure of a type but leave the type itself open. This data structure can be uncovered by considering abstract and non-abstract usage.

Non-abstract usage of a global variable G can be either use of G by a function outside of the cluster G belongs to or use of G by a function inside the cluster. Abstract usage is non-direct usage whereby a function uses G by calling an accessor routine of G. A non-abstract usage of a type is a direct access to an element of an instance of the type. For example, accessing a field of a record, an index subscript of an array, or dereferencing a pointer can all be considered as non-abstract usages. An abstract use of a type is manipulation of the type by a function through an accessor routine of the type.

It is desirable to recluster functions when there are non-abstract usages of types and global variables by functions that are outside of the cluster the types or variables belong to. This can be done by redefining *preSet* and *postSet* from their original definitions by Canfora [CANF96] for a component S and a function belonging to S, e:

$$preSet(S,e) \Leftrightarrow reference(S,e) \vee (type(S,e) \wedge non-abstract(S,e))$$

$$preSet(e) = \{S | preSet(S,e)\}$$

$$postSet(e) = \{e | postSet(S,e)\}$$

Here, *reference(S,e)* is the set of global variable references by e, *type(S,e)* is the set of type references by e and *non-abstract(S,e)* is the set of non-abstract usages of data items by e.

Koschke and Girard found that this revised approach does provide more appropriate clusters than the original Delta IC approach (described in Section 3.1.9) and will allow Delta IC to be used to program comprehension rather than restructuring; however, the user still has to define the Delta IC threshold, which can be time-consuming to establish and which will be different for each software system.

3.1.13.4 Similarity Clustering

The culmination of Koschke and Girard's work on automatic software clustering methods is a technique called *Similarity Clustering* ([GIRA99], [KOSC00a]), which is based on Schwanke's Arch approach [SCHW91]. Similarity Clustering develops the

metric presented by Schwanke and incorporates a wide range of descriptive features of a software system, many of which can be weighted and adjusted as the user sees fit.

Similarity Clustering uses a basic hierarchical algorithm, resulting in a dendrogram. Firstly, the similarity between each pair of entities is calculated based on a number of criteria. The clustering can then take place, with the similarity values between two clusters X and Y calculated using the following equation:

$$GSim(X,Y) = \frac{\sum_{(x_i \in X, y_j \in Y)} Sim(x_i, y_j)}{size(X) \times size(Y)}$$

The similarity value between a pair of entities A and B is calculated using the following overall equation:

$$Sim(A,B) = x_1 \cdot Sim_{indirect}(A,B) + x_2 \cdot Sim_{direct}(A,B) + x_3 \cdot Sim_{\inf ormal}(A,B)$$

In Koschke and Girard's original definition of Similarity Clustering [GIRA99], similarity values were normalised to have a value between 0 and 1. However, the later definition in Koschke's thesis [KOSC00a] does not normalise similarities. This is perhaps due to the fact that non-normalised values allow the number of times a feature occurs between two entities to contribute to the similarity value. The three parameters x_1, x_2 and x_3 govern the relevance of the three main parts of the equation. These parts, $Sim_{indirect}(A,B)$, $Sim_{direct}(A,B)$ and $Sim_{informal}(A,B)$, are now defined.

Direct relations, represented by $Sim_{direct}(A,B)$, are relations concerning A and B alone, represented by connections between A and B. These are defined by the following:

$$Sim_{direct}(A,B) = W(Link(A,B))$$

where $Link(A,B)$ represents the set direct links between A and B. The W signifies that the links in this set will be weighted; the precise method of weighting is described after the definition of $Sim_{indirect}(A,B)$.

Indirect relations, represented by $Sim_{indirect}(A,B)$, are relations between A and B via some third party. These are defined by the following:

$$Sim_{indirect}(A,B) = \frac{I_{eq} \times W(Common_{eq}(A,B)) + W(Common_{ne}(A,B))}{I_{eq} \times W(Common_{eq}(A,B)) + W(Common_{ne}(A,B)) + d \cdot W(Distinct(A,B))}$$

Here, $Common_{eq}(A,B)$ and $Common_{ne}(A,B)$ represent the common features of A and B. $Common_{eq}(A,B)$ is the set of equivalent features of A and B, where A and B's common

neighbours are used in the same way. For example, if A and B both return a value of the same type, they use this type feature in an equivalent fashion. $Common_{ne}(A,B)$ is the set of non-equivalent features of A and B, where A and B's common neighbours are used in different ways. This means if A returns a value of a certain type and B takes a variable of this type as a parameter, this type feature will be part of $Common_{ne}(A,B)$. I_{eq} determines the influence of equivalent features, which should be higher than the influence of non-equivalent features because a common use of a common neighbour is more significant than just a common neighbour that is related in different ways [KOSC00a].

$Distinct(A,B)$ represents distinct features and d represents the influence these features have. The inclusion of distinct features as a mark of similarity is not necessarily desirable, but its use can avoid finding two entities similar which share one common feature but are otherwise different, which may happen if only common features are considered.

The weighting of features uses Shannon information content, as Schwanke's Arch approach did [SCHW91]. The weight of a set of features (as used above) is defined as:

$$W(X) = \sum_{x \in X} weight(x)$$

The weight of an individual feature is calculated using a combined weight strategy. This involves taking into account the probability of the class of feature occurring within the particular software system being analysed. It also takes into account the number of features that describe the current entity. This is computationally expensive but provides a greater depth of description than any other software clustering technique.

Similarity Clustering also uses informal features in a more detailed way than any other software clustering technique [GIRA99]. Although it does not consider comments describing an entity to be features of the system, names of identifiers are considered valid features. These are included both as whole words and as common parts of words. Filenames used are also considered. $Sim_{informal}(A,B)$ is defined as follows:

$$Sim_{\text{inf ormal}}(A,B) = Sim_{words}(A,B) + Sim_{suffix}(A,B) + Sim_{filename}(A,B)$$

$Sim_{words}(A,B)$ is the set of common words of A and B. These words are taken from the names of identifiers within A and B. These identifiers are separated when underscores

are encountered (for example, *list_insert* counts as two words) or capital letters are used (*ListInsert* would also be separated into two).

Similar information is contained by the set $Sim_{suffix}(A,B)$, which is defined by the following:

$$Sim_{suffix}(X,Y) = \frac{prefix(X,Y) + postfix(X,Y)}{1 + prefix(X,Y) + postfix(X,Y)} \text{ when } X \neq Y \text{ ; 1 when } X=Y$$

Here, *prefix(X,Y)* and *postfix(X,Y)* are the lengths of the common pre- and postfix of their two arguments if the length is longer than three characters; otherwise they are zero.

$Sim_{filename}(A,B)$ uses $Sim_{suffix}(A,B)$ to match common bodies of filenames. For example, *list.c* and *list.h* may be grouped together based on the common prefix *list*.

$$Sim_{filename}(X,Y) = Sim_{suffix}(filename(X), filename(Y))$$

The similarity metric used for Similarity Clustering is clearly the most complex metric devised for software clustering and is computationally expensive. There are also a large number of parameters that must be altered by the user depending on the software system being analysed and the purpose of the analysis in order to use the metric successfully. Koschke provides a number of iterative methods for defining these parameters [KOSC00a], such as using a Simulated Annealing algorithm to refine an assumed parameter value, and also suggest some practical edge weightings that can be refined depending on the system being analysed.

3.2 Concept Analysis Techniques

This section describes in detail the basic method used to perform concept analysis and the representation that is often used to display sets of concepts, the concept lattice. The work done by individual researchers to improve the results provided by the basic concept analysis method is then discussed.

Concept analysis requires a description of the system based on entities and features. As with cluster analysis, the descriptions frequently use functions as entities and use of global variables, user-defined types and other functions as features [SIFF97]. Most current approaches, which have been tested on COBOL systems, only use global

variables to describe the system, but there is no reason why any of the features used by the software clustering techniques that have been described in this chapter couldn't be used for concept analysis.

Concept analysis descriptions are usually binary relations [LIND97], which are only able to record the presence or absence of a feature; the strength of a feature's presence (such as the number of times a global variable is used) cannot be included in the description. This weakens the quality of the description, especially for software where the repeated use of a variable or type strongly suggests that that variable or type should be clustered with the function using it.

	f1	f2	f3	f4	f5
e1	X	X			
e2		X	X	X	X
e3	X	X			
e4			X		
e5		X			X

Table 3.1: Example binary relation for concept analysis

Table 3.1 shows an example description of a software system containing five entities (*e1-e5*) and five features (*f1-f5*). If an entity uses a feature, an X is added in the appropriate place in the table. The table is, formally, a relation $T \subseteq E \times F$ where E is the set of entities and F is the set of features. From this relation, *concepts* can be defined. A concept is a collection of entities and features such that the entities in the concept share the features in the concept. This can be defined specifically by considering the sets E and F.

For a subset of entities A belonging to E, the set of common features of A is defined as:

$$\sigma(A) = \{f \in F \mid \forall e \in E : (e, f) \in T\}$$

For a subset of features B belonging to F, the set of common entities of B is defined as:

$$\tau(B) = \{e \in E \mid \forall f \in F : (e, f) \in T\}$$

A concept is a pair of sets (A,B) of entities and features such that $B = \sigma(A)$ and $A = \tau(B)$. The set of entities A is known as the *extent* of the concept and the set of features B is known as the *intent* of the concept.

top	$(\{e_1, e_2, e_3, e_4, e_5\}, \phi)$
c_0	$(\{e_1, e_2, e_3, e_5\}, \{f_2\})$
c_1	$(\{e_2, e_4\}, \{f_3\})$
c_2	$(\{e_2, e_5\}, \{f_2, f_5\})$
c_3	$(\{e_1, e_3\}, \{f_1, f_2\})$
c_4	$(\{e_2\}, \{f_2, f_3, f_4, f_5\})$
bottom	$(\phi, \{f_1, f_2, f_3, f_4, f_5\})$

Table 3.2: Example set of concepts

Table 3.2 shows the set of concepts that is created by analysing the binary relation in Table 3.1. This set of concepts can be represented using a *concept lattice* (see Figure 3.3), also known as a Galois lattice [SAHR99], which has a hierarchical structure similar to a dendrogram. Unlike a dendrogram, however, overlapping concepts can be represented in the lattice, meaning that no (possibly incorrect) decisions need to be made as to which entities and features fit with which concepts.

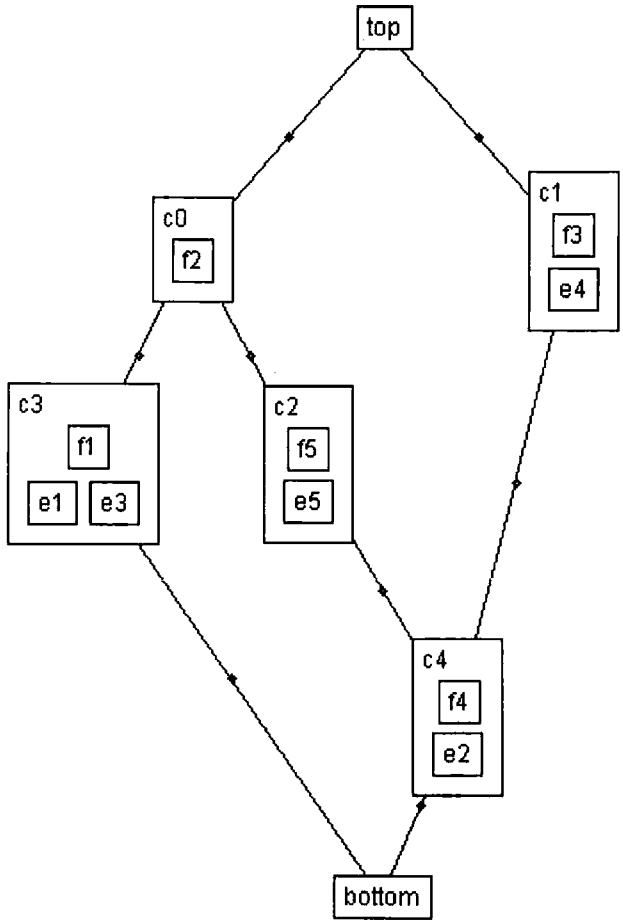


Figure 3.3: Concept lattice example

Each node in the lattice represents a concept (*c0* to *c4* in Figure 3.3). There are special concepts at the top and bottom of the lattice. The *top* node represents a concept

containing all entities but no features. The *bottom* node represents a concept containing no nodes but all features. The lattice is constructed so that, as the paths of the lattice progress downward, the extent of each concept node becomes smaller and the intent of each concept node becomes larger [DEUR99b].

Because each concept can have many elements, the nodes are labelled with a feature (*f1* to *f5* in Figure 3.3) if it represents the largest concept with that feature in its intent and with an entity (*e1* to *e5* in Figure 3.3) if it represents the smallest concept with that entity in its extent. This labelling strategy means that the lattice can be used to observe both which features are used by an entity and which entities share these features by traversing the lattice. It is also easy to identify disjoint sets of entities and features that could possibly form a module by isolating a set of concepts which form a path from the top node to the bottom node in the lattice.

However, the concept lattice still has the same problem as the dendrogram and other system representations, in that further analysis is required to produce a high level abstraction of the system. Although concepts themselves can be used to form abstract data objects, this is not ideal to create a complete abstraction because concepts can overlap; also, each concept may be too small or too large to form a suitable ADO. Concept analysis researchers have proposed various different ways of dealing with this problem.

Lindig and Snelting [LIND97], claimed that, if a concept lattice for a system is horizontally decomposable, the system could be modularised. A lattice is horizontally decomposable if it consists solely of independent substructures that are connected only via the top and bottom nodes.

This is very unlikely for a legacy system and so Lindig and Snelting also describe a method for dealing with interference. Two sets of entities are said to be *coupled* if they use the same features, and two sets of features are said to *interfere* if they are used by the same entities. Interferences must be dealt with in order to create suitable modules. A small number of interferences is permissible because the entities they affect can be placed in the same module. It may also be possible to perform some simple program transformations in order to destroy the interference, for example by defining an accessor function for a variable rather than accessing the variable directly.

However, if there are a very large number of interferences or the system itself is very large (or both, as is common for legacy systems) it is often not possible to deal with the interferences sufficiently in order to produce a horizontal decomposition of the lattice. It is also very difficult to comprehend such a lattice because of the large number of connections between concepts. In fact, the reverse engineering project where Lindig and Snelting attempted to use concept analysis had to be abandoned because the concept lattice proved too convoluted to provide any usable results [LIND97].

Siff and Reps [SIFF97] attempt to formalise the partitioning problem by defining a *concept partition*. A concept partition contains no overlapping concepts and is equivalent to a collection of modules such that every function in the program is associated with exactly one module. Two special partitions are defined; the *atomic partition*, which is where there is only one function in each module, and the *trivial partition*, where all functions are placed in the same module. These are equivalent to the first and last partitions in a dendrogram.

Siff and Reps claim that it is possible to discover all concept partitions in a concept lattice and that an atomic partition is a good starting point for discovering a good modularisation of the system. However, not all lattices will contain an atomic partition depending on how the concepts overlap so this may not be possible.

Unfortunately, Siff and Reps do not propose a method for choosing a good partition from the set of all partitions, the lack of which is still a problem for using dendrograms as well. This is a very significant problem, especially because the case study performed by Siff and Reps on a code sample of just one thousand lines produced 153 partitions of the concept lattice.

For actual industrial sized legacy systems it is thought that some automatic refinement of the concept lattice or the set of concept partitions must be included into the concept analysis process if it is to be usable. Canfora et al [CANF99] propose some simple refinements of the concept lattice, such as isolating highly connected functions or sublattices only connected to the rest of the lattice by the top and bottom nodes, but it is thought that the remaining lattice will still be very difficult to comprehend and partition.

In conclusion, the concept lattice suffers the same major problems as the dendrogram, in that they are both difficult to partition and understand both manually and automatically. The concept lattice has the distinct advantage that all possible concepts are included in the lattice and no modularisation decisions are made during its construction, unlike the dendrogram where bad decisions early in the clustering process terminally affects the rest of the process. However, the creation of the concept lattice is computationally expensive and it is unknown whether concept analysis techniques will scale up successfully to industrial legacy systems as opposed to the small case studies current concept analysis researchers have relied on.

3.3 Conclusion

Such a mass of different techniques, each with their own frame of reference and style, makes it very difficult to attempt a coherent assessment of software clustering and concept analysis techniques on any general terms. Luckily, in recent years, some general work has emerged that attempts to analyse, compare and classify these techniques. This work is described in the following chapter.

CHAPTER FOUR – Software Clustering Assessment

With the ever-increasing number of software clustering techniques, more general work has emerged in the field, either attempting to compare and classify existing techniques or to solve common problems of clustering methods. This work is concerned with software clustering techniques only; concept analysis techniques have not yet been considered in a general context. This chapter describes some of the work that has attempted to assess the field of software clustering.

4.1 General Surveys

Lakhotia [LAKH97] provided the first major survey of software clustering methods and also created a framework for classification. Various terminology, notation and symbols are defined to describe the inputs, outputs and processing of SCTs (subsystem classification techniques). This framework is used to reformulate many different SCTs, including some of the ones described in Chapter Three. The stated aim for the framework is to allow comparison of SCTs to ease the selection of an SCT for a particular purpose and to allow 'mixing and matching' of SCT elements to experiment with and create new SCTs.

Unfortunately, Lakhotia does not attempt any comparison of the SCTs that have been reformulated and, so far, no one else has used the framework to do so. This is possibly because the developed framework is difficult to understand, due to its use of many unfamiliar acronyms and symbols. Furthermore, because of the variety of software clustering techniques, it is still necessary to describe the techniques in a fair amount of detail even within the framework, and the terminology tends to make these descriptions more difficult to understand than the original authors' descriptions.

A more successful approach is taken by Wiggerts [WIGG97], who presents his approach to software clustering algorithms from the field of cluster analysis rather than reverse engineering. This view throws new light on many existing methods by looking at them conceptually rather than tied to a particular software paradigm or architecture. Although other researchers had already used methods developed in other disciplines,

Wiggerts was the first to highlight the link between traditional cluster analysis and software clustering. He defined the terms entity and feature as they are used here and specified three questions which must be answered to apply cluster analysis, which roughly correspond to the identification of *entities*, *clustering criterion* and *clustering algorithm* described in Section 2.2.2:

- What are the entities to be clustered?
- When are two entities to be found similar?
- What algorithm do we apply?

A basic classification of features, similarity metrics and clustering algorithms is also included, many of which had not been applied to software clustering when Wiggerts presented his work and some of which have still not been used in a reverse engineering context.

Wiggerts's work was developed by Anquetil and Lethbridge [ANQU99], who attempted to use a number of the methods Wiggerts described for reverse engineering purposes. Only metric-based, hierarchical approaches were assessed, which limits the usefulness of the survey. However, there is a good description of the possible features that can be used to describe the possible entities of a system.

In particular, Anquetil and Lethbridge make a good case for the use of informal information such as identifiers or comments in the code for clustering, if used in conjunction with more formal features. They note the need to combine features to construct a useful view of the system, but claim that there is much redundancy between formal features (for example, reference to a global variable implies there is a reference to the type of that variable) and the low-level nature of formal features makes it difficult to extract abstract concepts. Informal information can be used to better recognise the semantics of a system and, by doing so, decrease the redundancy of the system representation by more strongly identifying syntactically similar entities.

4.2 Tzerpos & Holt

Tzerpos and Holt were among the first researchers to propose methods to assess the representations produced by software clustering techniques, as well as providing a general framework for software clustering techniques. Their work is now described.

4.2.1 Framework and Comparison

Possibly the most useful survey of software clustering methods is 'Software Botryology' (see Section 2.2.2) by Tzerpos and Holt [TZER98]. This provides a succinct description of some of the major software clustering approaches and a framework for a generalised software clustering technique, which is described (adapted) as follows:

1. *Data collection* – relevant information is extracted from source code
2. *Initial screening* – the data is massaged to suit the purpose of the technique (for example, removal of library functions)
3. *Representation* – an appropriate similarity measure is chosen
4. *Clustering strategy* – a clustering algorithm is chosen and any parameters needed to run the algorithm are set
5. *Validation* – the resulting clusters are validated, usually by the maintainer
6. *Interpretation* – the results are compared with other studies in an attempt to improve the clusters and clustering process

Tzerpos and Holt also list some of the major research issues with software clustering techniques, including which features, similarity measures and clustering algorithms are appropriate for software clustering and also the need to test software clustering methods on large systems and dynamic systems. Many of the problems they list have been solved to some extent by the work of Koschke and Girard (see Section 4.3). However, Tzerpos and Holt have done much interesting further work on the use of software clustering methods over a period of time, rather than as a single instance preventative maintenance measure.

Tzerpos and Holt's subsequent work has focussed on ways to compare software clustering algorithms and their results, which has led to the definition of *MoJo*, a distance measure which can be used to compare two clusterings of a software system [TZER99]. Three reasons are given for the definition of the MoJo metric. Firstly, many

existing software clustering methods incorporate parameters that are used to fine-tune the results of the method; MoJo can be used to assess the influence of these parameters. Secondly, where an expert clustering of the system exists, MoJo can be used to assess the results of software clustering methods against it. Finally, the stability of software clustering algorithms can be assessed by using MoJo on clusterings of different versions of a system.

Tzerpos and Holt have used MoJo in further work on the stability of software clustering algorithms [TZER00a]. Working from the point of view that software clustering is more appropriate for program comprehension than restructuring, it is thought that the results produced by software clustering algorithms should keep the same basic structure if only a small amount of the system is changed, because this aids the comprehension of the clustering results.

Therefore, Tzerpos and Holt tested a number of hierarchical algorithms on various systems to assess their stability using MoJo. This work suggests that MoJo is appropriate for assessing stability but that a wider range of algorithms need to be tested to claim this with any certainty. As MoJo has been used to test a number of algorithms for this thesis, a full description of the algorithm used to calculate MoJo is now provided.

4.2.2 MoJo Description

MoJo is a comparison between two clusterings of a software system; more specifically, it represents the distance between two partitions of the same set of entities. This distance is the minimum number of operations needed to transform one partition into the other. Two operations are allowed: *move* and *join*. Each operation has a weight of 1. The smaller the number of *move* and *join* operations required to transform one partition into the other, the more similar the partitions are.

A *move* operation consists of moving an entity from one cluster to another. This includes moving an entity to a new cluster; creating the new cluster incurs no additional cost.

A *join* operation joins two clusters together. Because there is only one way to join two clusters together, the operation has a weight of 1.

Because there are many ways to split a cluster in two (as the entities can be moved with different orders), splitting a cluster into two separate clusters must be considered a series of *move* operations.

This difference between splitting and joining means that the minimum number of operations (mno) needed to transform a partition A into a partition B is not necessarily the same as the minimum number of operations needed to transform B into A. Tzerpos and Holt therefore define MoJo(A,B), where A and B are two partitions of the same set of entities S, as:

$$MoJo(A,B) = \min (mno(A,B), mno(B,A))$$

The set S must consist of distinct clusters of equal stature. For example, if a dendrogram has been produced by the clustering method, S must consist of a single partition of the dendrogram and cannot include clusters from other partitions.

Because there are many different ways to transform one partition into another, it can be very time-consuming to calculate the minimum number of operations needed to perform the transformation, especially for large sets of entities. Therefore Tzerpos and Holt propose a heuristic algorithm that calculates an approximation of the value of MoJo [TZER99].

Tzerpos and Holt claim that a MoJo value calculated for representations of two slightly different software systems produced by a clustering algorithm should be roughly the same as the amount of change between the software systems themselves. In Tzerpos and Holt's original experiments with the MoJo distance metric, a MoJo value was calculated between a code sample and that same code sample with 1% of the system altered. This experiment was repeated 100 times. Tzerpos and Holt found that the MoJo value calculated was less than 1% of the total size of the system in over 80% of the experiments, which suggests that MoJo values broadly correlate with the amount of change in the system the MoJo values are calculated on.

4.2.3 MoJo Algorithm

The following strategy is used to calculate an approximation of the minimum number of operations needed to transform a cluster partition A into a cluster partition B (mno(A,B)). Let A_i, $1 \le i \le l$, be the clusters of partition A and B_j, $1 \le j \le m$, be the

clusters of partition B. In order to limit the number of moves to be performed, it is necessary to keep as many entities in each cluster in partition A as possible. Therefore, each cluster B_j in partition B is assigned a *centre cluster* from partition A.

The *centre cluster* of a cluster B_j (denoted CC_j) is the cluster A_i that contains more entities from B_j than any other cluster in A. Essentially, CC_j is the most similar cluster in partition A to B_j. The aim of the algorithm is to convert each CC_j into B_j.

A simple strategy to accomplish this would be to move each entity that is in B_j and not in CC_j from its current cluster in A to CC_j. However, it is possible that, by joining certain clusters together, some of these moves could be avoided. Therefore, there are two main steps to the algorithm; calculate the set of centre clusters and look for profitable joins. These steps are now described.

4.2.3.1 Calculating centre clusters

To calculate the set of centre clusters, it is first necessary to tag the entities in partition A with their corresponding entities in partition B. Each entity e in partition A is assigned a tag T_j, $1 \leq j \leq m$, such that $T_j = B_j$, where B_j is the cluster that entity e belongs to in partition B. $A_i(T_j)$ is now defined as the set of entities in cluster A_i which are tagged T_j, and therefore can also be found in cluster B_j. This tagging means that it is only necessary to consider the tags in the clusters of partition A to calculate MoJo.

Centre clusters can now be assigned to each cluster B_j. CC_j is a cluster A_i in A such that $|A_i(T_j)| \geq |A_k(T_j)|, 1 \leq k \leq l$. It is possible that more than one cluster could fit this description; if this is the case, one of these suitable clusters is chosen arbitrarily.

Because the aim is to transform each centre cluster CC_j into cluster B_j, it is essential that each cluster A_i is centre cluster to only one cluster B_j. If there is a situation where one cluster A_z is centre cluster for two clusters B_x and B_y in partition B so that $CC_x = CC_y = A_z$, action must be taken to ensure that B_x and B_y in partition B have different centre clusters. There are two ways to do this.

Firstly, it may be possible to find other suitable centre clusters for B_x or B_y. Because $CC_x = CC_y = A_z$, A_z must contain the largest sets $A_i(T_x)$ and $A_i(T_y)$ for any cluster A_i. Therefore, the clusters in partition A containing the second largest sets $A_i(T_x)$ and $A_i(T_y)$

are the next candidates for replacing A_z as CC_x or CC_y. These clusters can be established by repeating the following procedure until all clusters in partition B have unique centre clusters or until no suitable changes can be made.

Before the procedure begins, sets $\Phi_k, 1 \leq k \leq m$ are defined, initially empty, to act as markers. These sets ensure no cluster A_i is considered more than once.

If there remain clusters B_x or B_y in partition B such that $CC_x = CC_y = A_z$ for some cluster A_z in partition A, the following values are defined:

$$x_1 = |A_z(T_x)| \qquad y_1 = |A_z(T_y)|$$

$$x_2 = |A_f(T_x)|, \text{ such that } |A_f(T_x)| \geq |A_i(T_x)|, 1 \leq i \leq k, i \neq z, i \notin \Phi_x$$

$$y_2 = |A_g(T_y)|, \text{ such that } |A_g(T_y)| \geq |A_j(T_y)|, 1 \leq j \leq k, j \neq z, i \notin \Phi_y$$

x_1 and y_1 are the sizes of sets of entities shared by B_x & A_z and B_y & A_z respectively. These sets are the largest such sets for all $|A_i(T_x)|, 1 \leq i \leq k$. x_2 and y_2 are the second largest of these sets, excluding entities that have already been considered, which are contained in the sets Φ_x and Φ_y.

If both $x_2 = 0$ and $y_2 = 0$, B_x and B_y share no similarity with any other cluster A_z. Therefore, no other cluster A_z can be considered to be the centre cluster of either B_x or B_y.

If only $x_2 = 0$, then only B_y has any similarity to another cluster A_z. This cluster, A_g, is made the centre cluster of B_y ($CC_y = A_g$) and z is added to Φ_y to ensure that it is not considered again.

If only $y_2 = 0$, then only B_x has any similarity to another cluster A_z. This cluster, A_f, is made the centre cluster of B_x ($CC_x = A_f$) and z is added to Φ_x to ensure it is not considered again.

If both $x_2 > 0$ and $y_2 > 0$, then both A_f and A_g may be suitable alternatives for A_x. If $x_1 + y_2 > y_1 + x_2$, $CC_y = A_g$ and z is added to Φ_y. Otherwise, $CC_x = A_f$ and z is added to Φ_x.

This is because it is important to minimise the moves and joins necessary, so the candidate centre cluster that causes the smallest change will be the new centre cluster.

The above procedure now repeats until as many centre clusters are made unique as possible. However, there is still a chance that there could still be situations where $CC_x = CC_y = A_z$, even after this procedure has been completed. For example, it will always happen if there are less clusters in partition A than in partition B. Therefore, these persistent centre clusters must be split into two, such that one of the new clusters contains the entities tagged T_x, such that $|A_z(T_x)| \leq |A_z(T_y)|$. It is necessary that the smaller of the two sets is taken because the split takes the form of a series of move operations that must be added to the total value of *mno(A,B)*.

4.2.3.2 Seeking profitable joins

As stated before, once the central clusters are established, it is possible to move every entity tagged T_j into CC_j. However, because joining two clusters has a weight of 1, the same as moving an entity from one cluster to another, it may be more efficient for some pairs of clusters to join them together and move a smaller number of entities than would have been moved without the join.

For two clusters A_i and A_j, where A_i is a centre cluster CC_x, the completed join would involve joining the clusters and then moving all entities not tagged with x into the appropriate clusters. There are two cases to consider in order to establish whether this is worthwhile.

If A_j is not a centre cluster, it is profitable to join A_i and A_j providing $A_i(T_x) > 1$, because if this join is not made, it will cost at least $|A_j(T_x)|$ to move the set $A_j(T_x)$ into A_i.

If A_j is a centre cluster CC_y, then the following values must be calculated to establish whether the join is profitable or not:

$$A = |CC_x| \qquad \Gamma = |CC_y|$$

$$\alpha = |CC_x(T_x)| \qquad \beta = |CC_y(T_x)|$$

$$\gamma = |CC_y(T_y)| \qquad \delta = |CC_x(T_y)|$$

Firstly, it is important that $\beta \geq \delta$. If CC_x and CC_y are not joined, these values represent the number of moves between CC_x and CC_y that will have to take place. However, even if the join does take place, the set $CC_x(T_y)$ will have to be moved back into CC_y. Therefore, if $\beta < \delta$, more entities will be moved than necessary, so the figures must be recalculated with A_i as CC_y and A_j as CC_x.

$A-\alpha$ is the number of entities that are not in B_x and therefore do not belong in CC_x, as $\Gamma-\gamma$ is the number of entities that do not belong in CC_y. Therefore, to transform CC_x into B_x and CC_y and B_y using move operations alone will incur a cost of $A-\alpha+\Gamma-\gamma$. Profitably joining CC_x and CC_y will incur a cost of 1 for the actual join, followed by a cost of $A-\alpha$ for the entities in CC_x that are not in B_x, plus a cost of $\Gamma-\beta$ for all the entities that were originally in CC_y that now need to be moved back into CC_y or moved elsewhere. This means such a join is profitable only if:

$$A-\alpha+\Gamma-\gamma > 1+A-\alpha+\Gamma-\beta$$

$$-\gamma > 1-\beta$$

$$\beta > 1+\gamma$$

Therefore, if $\beta > 1+\gamma$ the join is profitable and the two clusters A_i and A_j are joined to form a new cluster A_k at cost 1. When the move operations are performed later, each of the entities marked T_y will be moved back into A_j and everything else moved to the appropriate clusters; the cost should still be less than if the set $A_i(T_x)$ had been moved, one by one.

All possible pairs of clusters must be examined to see if joining them would be profitable. In order to maximise performance, the clusters C in partition A are ordered by the sets $C(T_i)$ such that if $C=CC_i$, $i \neq j$. The process of checking for joins then begins with the two clusters with the largest such sets. Note that if two clusters are joined, this process must restart with the order revised, as the set of clusters or the entities' tags will have changed.

Once all profitable joins have been performed, the remaining entities tagged T_j must be moved into CC_j, increasing the value of *mno(A,B)*. The whole process must then be repeated for *mno(B,A)* and the smaller of the two values is taken as the approximation of *MoJo(A,B)*.

The MoJo algorithm has been adapted for this thesis. The adaptations and the reasons for their implementation are discussed in Section 5.4.

4.3 Bauhaus

In recent years, the most comprehensive work in software clustering has come from the Bauhaus research group, in particular Koschke and Girard. Their work has taken up the challenges presented by Wiggerts [WIGG97], Lakhotia [LAKH97] and Tzerpos and Holt [TZER98] through extensive surveys and adaptations of previous clustering methods, including detailed reworkings of Schwanke's Arch approach [SCHW91] and Canfora et al's Delta IC approach [CANF96]. The adaptations of Arch and Delta IC, as well as two new approaches, are discussed in Section 3.1.13. The current section describes the classification and comparison of software clustering techniques performed by Koschke and Girard and draws some observations based on this work that have led to the survey described in the remainder of this thesis.

In order to complete the assessment work presented in this section, the Bauhaus team have implemented a tool suite known as Bauhaus Rigi. This suite, which is an extended version of the Rigi editor [MÜLL93] discussed in Section 3.1.6, provides a complete process for component recovery. Almost all of the software clustering and concept analysis techniques described in Chapter Three have been implemented as part of Bauhaus Rigi. Any ANSI C code can be used as input to the Bauhaus parser, which produces base graphical representations of the entities in the code. Any or all of the implemented techniques can then be used to analyse the system and the results of these techniques can be examined and manipulated individually or combined as appropriate. This tool suite has also been used to assess the techniques examined for this thesis; more information can be found on these techniques in Section 5.2.

4.3.1 Classification

Based on his study of a wide range of software clustering techniques, Koschke provides a basic classification of what he terms 'structured component recovery techniques' [KOSC00a]. He defines four main branches of technique that are based on structural information such as calling structures and type usage and renames many existing techniques. The branches are:

- **Connection-based** approaches, which cluster entities based on a specific set of direct relationships between entities to be grouped. Liu and Wilde's work [LIU90] (Section 3.1.4), renamed as Part Type by Koschke, Same Module [GIRA00] and Same Expression [KOSC00a] (both Section 3.1.13) are examples of this type of technique.
- **Metric-based** approaches, which cluster entities based on a metric using an iterative clustering approach. Similarity Clustering [GIRA99] (Section 3.1.13) is a metric-based approach and Koschke also classifies Delta IC [CANF96, CANF00] (Sections 3.1.9 and 3.1.13) as metric-based with some reservations, as the Delta IC metric only forms part of that technique's clustering process.
- **Graph-based** approaches, where clusters are derived from a graph by means of graph-theoretic analyses. These differ from connection-based approaches because a whole graph must be considered, rather than only direct relationships between entities. Cimitile and Visaggio's Dominance Analysis approach [CIMI95] (Section 3.1.10) is a graph-based approach.
- **Concept-based** approaches, where concept analysis is used to create a concept lattice based on a binary relation derived from a software system. This includes the work described in Section 3.2 by Lindig & Snelting [LIND97] and Siff & Reps [SIFF97].

This classification has been used to select the techniques analysed for this thesis.

4.3.2 Comparison

In order to compare existing software clustering techniques, Koschke and Eisenbarth defined an evaluation framework [KOSC00b], taking their lead from the ideas of Lakhotia [LAKH97] (see Section 4.1). This framework relies on the comparison of candidate clusters to established reference clusters, identified as benchmarks by experienced software engineers. The framework was used to assess a wide range of software clustering techniques including Part Type, Same Module, Same Expression, Arch, Similarity Clustering and Delta IC, among others.

A foundation of this approach is the recognition that, although a set of clusters may not be a perfect representation of the system, they may be good enough for a desired purpose. Therefore, two relationships are defined: an *affinity* relationship, which establishes to what extent two clusters overlap, and a *partial subset* relationship, which

determines whether a cluster is similar to only part of another cluster. The relationships for two clusters C and R are described as follows:

Affinity: $C \approx_p R$ if and only if $overlap(C,R) \geq p$

Partial subset: $C \subseteq_p R$ if and only if $\frac{|elements(C) \cap elements(R)|}{|elements(C)|} \geq p$

where

$$overlap(C,R) = \frac{|elements(C) \cap elements(R)|}{|elements(C) \cup elements(R)|}$$

Elements(X) denotes the set of base entities used by cluster X. The threshold value p allows the user to define how similar they wish the two clusters to be. A value of 1 would mean the two clusters are identical. Koschke and Eisenbarth suggest a value of 0.7 as a fair default value, meaning the clusters must share three out of four entities to be considered affine [KOSC00b]. Three types of matches are described for a candidate cluster C and a reference cluster R:

- **1~1:** true when $C \approx_p R$ (candidate is close to reference)
- **n~1:** true when $C \subseteq_p R$ (candidate is too detailed)
- **1~n:** true when $R \subseteq_p C$ (candidate is too large)

These matchings are used to calculate the detection quality of a software clustering technique. This quality is based on a number of values:

- Number of false positives and true negatives
- Average accuracies and level of granularity
- Recall rate

The number of false positives is the number of candidate clusters that cannot be associated with any reference cluster. The number of true negatives is the reverse of this: the number of reference clusters that cannot be associated with any candidate cluster. There should be no false positives or true negatives in a resulting representation and, if any of them remain after automatic analysis, the user of the system must remove them.

The level of granularity is assessed by examining the number of 1~1, n~1 and 1~n matches. A representation at an appropriate granularity level should have only 1~1 matches.

An accuracy factor is associated with each match in order to indicate the quality of imperfect matches of candidate and reference clusters. The average accuracies are calculated based on the overlapping of clusters, taking into account the size and matching type of the clusters. These accuracy values can then be used to calculate the recall rate of a technique, where *GOOD* is the set of resulting 1~1 matches, *OK* is the set of resulting 1~n and n~1 matches and *true_negatives* is the set of true negatives uncovered by the technique:

$$\text{Recall} = \frac{\sum_{(a,b)\in GOOD} accuracy(a,b) + \sum_{(a,b)\in OK} accuracy(a,b)}{|GOOD| + |OK| + |true\_negatives|}$$

Four medium-sized (30-40 KLOC) software systems written in C were used as a reference corpus. The reference components were created by three expert software engineers, who then reviewed the resulting component sets and selected the most appropriate components. The experts identified ADOs, ADTs and hybrid components in the systems but not related subprograms [KOSC00b].

A number of assumptions were made in order to perform the comparison. The techniques chosen often produce clusters with three elements or less; because such clusters are very rare in the expert clusterings, they are filtered out of the candidate clusterings. Clusters with more than 75 elements were also excluded. The semi-automatic elements of approaches such as Delta IC and Similarity Clustering were ignored, so no calibration of the parameters took place; only a reasonable estimate was used.

The results acquired by Koschke and Girard by performing this evaluation are now discussed and some observations based on these results that led to the work presented in the remainder of this thesis are provided.

4.3.3 Results and Observations

The results of the evaluation [GIRA00, KOSC00a] show that current automatic techniques do not even approach the success of human analysis. In most cases, the recall rate of the methods was between only 20 and 40 percent and the number of false positives and true negatives was high. However, when the false positives of the resulting candidates were analysed, it emerged that over 40 percent of them could be considered correct positives that had been overlooked in the manual analysis. Also, some common patterns emerged within the remaining false positives that could be used to filter out inaccurately defined components. For example, variables that are defined as global but are actually treated as local constants should not be considered a base for an ADO.

The system being analysed affected the results achieved greatly. Some of the poor results are due to the fact that the analysed system has an incredibly convoluted structure and only a certain amount of this structure can be organised meaningfully through automatic analysis or otherwise. This observation might suggest that the basis for performing automatic software clustering is flawed from the outset, as the techniques are precisely aimed at such convoluted systems. However, the following observations of the evaluation method itself suggest that the investigation of these techniques is not so futile.

It was found that Same Module recalled more ADOs and ADTs than any other technique for all but one system (although the margin was small) [KOSC00a]. This may seem strange for such a simple heuristic, compared to the complexities and subtleties of Delta IC and Arch. However, Same Module only uncovers the structure defined by the programmers of the system when the file structure was created. When the reference corpus was created, the expert software engineers would have seen this file structure and incorporated this representation into their mental model of the system. Also, any variable or function names will have been incorporated into this model.

Even if an attempt is made to ignore any semantic information, it is thought that it must have some impact on any manual analysis. This information is not necessarily correct, as the programmers may have been incorrect to place a set of functions in the same file or to name variables and functions according to a certain scheme, but is likely to mirror the results of the Same Module heuristic more closely than other methods. This problem

is especially evident in legacy systems, where any attempted structure placed on files or naming conventions will almost certainly have been broken by prolonged maintenance of the system.

Therefore, it is still believed that there is a place for automatic software clustering techniques, especially ones that rely on the actual, rather than perceived, structure of the system. However, the results are still poor, even accounting for the flaws in the evaluation method, and this suggests that proposing full restructurings of the system, let alone reuse candidates, is out of the question.

In light of this, the work presented in this thesis focuses on the use of software clustering techniques to study the evolution of software, where the use of the techniques involves program comprehension, not restructuring. The representations created by these methods provide the maintainers with a fresh picture of the system that may challenge their existing mental model of the system and help them to approach maintenance in a more productive manner. This outcome is suggested by the fact that many of the false positives uncovered by the techniques actually turned out to be correct positives. Also, it took around 20 to 35 hours (the best part of a working week) to produce a set of components manually [GIRA00]; automatic techniques rarely take more than a matter of minutes to produce their results. Even though further analysis will certainly be required, the wealth of useful information that automatic techniques could provide cannot be ignored.

It is thought that software clustering techniques may be able to uncover evolutionary patterns. By their nature, these methods are good at extracting the cohesive components of a system and less definitive when the components of a system are very disjointed. Therefore, they could be used to track the degeneration of originally cohesive components due to prolonged maintenance. This information could then be used to not only make existing components more cohesive but also to pinpoint and reverse undesirable trends in the maintenance of the system.

4.4 Conclusion

This chapter has explored some general research concerning software clustering and concept analysis techniques. The motivation for the work carried out for this thesis has also been explained. The following chapter describes this work in more detail.

CHAPTER FIVE - Analysis

This chapter describes the work undertaken to assess whether software clustering and concept analysis techniques can be used to assist software evolution. The methods of assessment and the techniques that were tested are listed and justified and the test software system is described. The use and adaptations of the MoJo distance metric for the current analysis are explained in detail. Finally, some of the tools developed to support the production and analysis of results are described.

5.1 Methods of Analysis

The work described here attempts to determine whether software clustering and concept analysis techniques can have any application in the context of software evolution. Because such analysis has not previously been undertaken, the aim is simply to confirm that such an application is possible and to make suitable recommendations for future work should such an application be desired, rather than to propose a well-defined process for doing so.

As described in Section 4.3.3, it is theorised that the techniques analysed here will provide results that can be used to demonstrate the changing evolutionary nature of a software system. The techniques will be run on consecutive versions of a software system and each will provide a representation of each version of the system. A maintainer of the system could examine the results to see how the use of a function, data type, variable or any other entity has changed over the development of the system. The change in structure of a module of the system could also be followed. This information could be used to assess the degree of legacy tendencies of the system and perhaps suggest the nature of some necessary preventative maintenance.

For this to be possible, the results must exhibit a number of properties. Firstly, each view of the system must contain as many entities (such as functions, variables and types) as possible in order to present as complete a representation as possible. This is essential if these views are to be used to inform a maintenance decision. Unfortunately, as explained in Chapter Four, no clustering technique includes complete knowledge of a

system. Therefore, it is important to assess whether the techniques tested include enough information to make them even partially useful to a maintainer.

Secondly, if the views acquired for different versions of a software system are to be compared, these views must remain relatively stable. If a large percentage of the view changes when there is only a small change from version to version, it will be almost impossible to detect any evolutionary trends over the development of the system and difficult to follow the progress of a single entity.

Therefore, both the coverage and the stability of the software clustering and concept analysis techniques assessed will be examined. The coverage of each technique will be established by comparing the number of entities of various types covered by the techniques as percentages of the total number of entities in the system. The stability of each technique will be assessed using the MoJo distance metric, explained in full in Section 4.2.2.

To add to this experimental evidence, a case study focussing on one particular group of functions in the system, representing around ten percent of the total number of functions, has been carried out. Section 5.3 describes the nature of this case study. It is hoped that this study will demonstrate how these techniques may be used in a maintenance situation and highlight some key issues concerning the appropriateness of these techniques.

The techniques that were assessed for this project are now discussed.

5.2 Chosen Techniques

Koschke's classification of component recovery techniques [KOSC00a], outlined in Section 4.3.1, defines four types of technique: connection-based, metric-based, graph-based and concept-based. Based on this classification, one technique of each type has been assessed. This is not to suggest that one technique will be completely representative of the whole class of techniques; however, it is thought that a sufficient number of general observations about the nature of each class will be possible to make this a worthwhile strategy.

The chosen software clustering techniques, using Koschke's names where applicable, are:

i) Part Type (connection-based), the second of Liu and Wilde's clustering methods described in Section 3.1.4 [LIU90]

ii) Dominance Analysis (graph-based) by Cimitile and Visaggio, described in Section 3.1.10 [CIMI95]

iii) Similarity Clustering (metric-based), the Bauhaus group's method described in Section 3.1.13.4 [KOSC00a]

iv) Concept Analysis (concept-based) by Lindig and Snelting, which follows the standard concept analysis procedure described in Section 3.2 [LIND97]

These techniques have been implemented as part of Bauhaus Rigi, the tool suite described in Section 4.3. The four techniques listed above were chosen because they provided the best, most understandable results in Bauhaus Rigi for the techniques in their class. Some manipulation was required to generate the results used in the form of a set of single level clusters, each containing a set of entities. Part Type is the only method where the raw results were in this form.

The Dominance Analysis results took the form of the dominance tree described in Section 3.1.10; however, as Koschke's implementation was used to produce the trees, variables and types were included in the analysis as well as functions. This tree was clustered by placing each first-level branch of the tree into a cluster (so that everything connected to the root of the tree belongs to a cluster).

As explained in Section 3.1.13.4, Similarity Clustering can be manipulated using a number of parameters. The default parameters were used for the current analyses, on the assumption that the average maintainer, working under pressure, would use the default parameters rather than spend a considerable amount of time refining the input to the process. Only one change was made; by default, informal information (such as comments and entity names) is not included in the clustering process. For this analysis, informal information was included, but formal information was given five times the weight of informal information.

The preliminary result of Similarity Clustering is a dendrogram, as described in Section 3.1.2. Bauhaus Rigi allows the user to partition dendrograms at a particular cut-off point, which represents a particular similarity value. After a number of experiments, a value of 0.2 was chosen for this analysis, meaning that no more clustering was performed after the greatest similarity between any pair of clusters dropped below 0.2.

Although Linding and Snelting's basic concept analysis approach was used, no attempt was made to partition the concept lattice this approach created because, as suggested in Section 3.2, there were too many interferences in the lattices to create a suitable partition. Therefore, the sets of concepts produced for each version were used as representations.

5.3 Analysed Software

This section describes the software system that the techniques described in Section 5.2 produced representations for. The part of the system used for the case study mentioned in Section 5.1 is also described. This system is still available and so all specific details about the system have been changed to protect its commercial identity.

5.3.1 System description

The system forms the database component of a larger industrial product. There are currently four versions of the system available, each composed of various C files. The largest file contains all the functionality of the system and the remaining files define some of the types used by this main source file. The system interacts with various library functions and some files from other parts of the product (again, mostly to import data types). The system is therefore complex enough to provide difficulties for the clustering techniques being assessed but is also small enough to allow manual examination and comparison of the results.

The whole product has a convoluted history. The first version was developed in mainland Europe and was an adaptation of an earlier, similar system. This version is designed to run on the OS/2 platform. Many of the comments and some of the entity names are not in English. The second version was also developed in mainland Europe and, along with numerous other enhancements, the code was adapted to run on the

Windows NT platform as well as OS/2. After the second version was shipped, the development of the product was transferred to England.

The third version of the database component shows little change as development was focussed on other components of the product; the changes that were made are mostly bug fixes or adaptations to comply with changes in other components. The fourth and current version, however, shows a large amount of change; some major enhancements were implemented but most importantly some preventative maintenance was carried out on the component. As well as some structural refinement and clarification, the comments and entity names were translated into English.

Table 5.1 provides some statistical information about the system described above. Within Table 5.1, System Entities are entities in other components of the whole product that are used by the database component. Library entities are entities from the standard C libraries that are used by the database component. The table clearly shows the development activity between Versions One and Two and Versions Three and Four, especially when compared to the very slight change between Versions Two and Three.

	Version One	**Version Two**	**Version Three**	**Version Four**
Lines Of Code	12,349	18,186	18,423	21,917
Functions	113	133	134	145
Variables	195	215	216	245
Constants	2	0	0	0
Types	101	83	86	98
System Entities	6	63	63	59
Library Entities	72	86	86	90

Table 5.1: Software system details

5.3.2 Case study

A large number of explanatory comments are provided at the start of the main source file for each version. These comments include a list of the functions in the code, grouped by the common functionality they implement in the view of the programmers. One of these groups has been chosen to be the focus of a case study. This group will be tracked through the views of the four versions generated by the four techniques, as it

might be by maintainers of a system if they wanted to make a change to one or more of the functions in the group.

The identified group of functions from the database component manage the files that contain the information in the database. The functions form an ADT centred on the data type DATABASEFILE; a variable of this type is created for each data file and each one contains information about the number of records in the file, how often the file has been accessed and other similar details. In the commented list of functions, this group remains unchanged throughout the four versions, suggesting either that the maintainers of the system believe this common functionality remains encapsulated and has not been altered or simply that this documented list has not been updated.

There are twelve functions in the case study group: *calcchecksum*, *datafileclose*, *datafilecreate*, *datafileopen*, *headerwrite*, *recordadd*, *recordcheck*, *recorddelete*, *recordread*, *recordmark*, *recordnew* and *recordwrite*. *recordadd*, *recordcheck*, *recorddelete*, *recordread*, *recordmark*, *recordnew* and *recordwrite* are concerned with the files in the database; *datafileclose*, *datafilecreate*, *datafileopen* and *headerwrite* are concerned with the files themselves. *calcchecksum* calculates a checksum for each file so that it can be ensured that the files have been written to correctly.

A full discussion of how this group of functions is treated by the four assessed techniques and the implications of this treatment for the techniques and the code itself can be found in Section 6.3.

5.4 MoJo Adaptations

As explained in Section 5.1, the stability of each of the analysed techniques must be assessed to establish if they can ever be used to examine evolution of software. The MoJo distance metric for software clusterings, developed by Tzerpos and Holt [TZER99] and described in Section 4.2.2, has been used to assess the stability of the analysed techniques. However, some adaptations of the metric are necessary for its use during this project. These adaptations and the reasons behind them are now explained.

MoJo is designed for use on two partitions of the same set of entities with no entity appearing more than once in each partition. However, the partitions that have been used as input for clustering techniques are based on different versions of a software system and will therefore contain different entities. Furthermore, the set of concepts produced by Concept Analysis includes multiple copies of entities, as each entity may appear in more than one concept. It is necessary to deal with both of these problems in order to make the MoJo values meaningful.

The problem of having two unequal partitions is dealt with by simply ignoring entities that do not appear in both partitions when calculating MoJo. The number of entities that only appear in the first partition and the number of entities that only appear in the second partition will be totalled and be compared to the MoJo results acquired. It is still valid to calculate MoJo for the entities that the two partitions share because it is still desirable that this set of entities should remain stable despite the introduction of new entities. The number of extra entities in both partitions can be used in part to judge whether the MoJo value shows that the clustering technique is suitably stable. For example, a high MoJo value might be explained by the fact that there were a large number of new entities introduced between two versions.

Calculating MoJo values for sets of concepts is more difficult. Each entity in each set may appear in more than one concept. If this is the case, it is impossible to tell which entity in the first concept set corresponds to which entity in the second concept set with any degree of certainty because there will be more than one possible match. This means it is not possible to tag the entities in the first set of concepts and so the set of centre clusters (in fact, centre concepts) can not be calculated using the method described in Section 4.2.3.1.

Therefore, the set of centre concepts is discovered by calculating the similarities of each concept in concept set A to each concept in concept set B. Concept B_j's centre concept is the concept A_i that is most similar to B_j.

Each similarity is calculated by counting the number of entities in A_i that are also in B_j and the number of entities in B_j that are also in A_i. These values are converted into percentages of the size of A_i and B_j respectively and an average is taken of the two percentages. This average is taken as the similarity between A_i and B_j.

Once all the similarities have been calculated, they are sorted for each concept, beginning with the greatest similarity. Centre concepts are then assigned by finding the concept A_i and the concept B_j with the greatest similarity and making A_i B_j's centre concept. This procedure repeats until all concepts in partition B have centre concepts.

The number of moves necessary to convert each centre concept into the corresponding concept in partition B can now be calculated. A check for profitable joins was done manually when the MoJo results for this thesis were generated based on these numbers of moves, but none were found to be necessary.

As with the other clustering methods, the MoJo values for the concept sets were only calculated for the set of common entities between the two partitions. Similarly, the number of unconsidered entities is also recorded. Once again, however, arriving at this figure is more complicated than it was for the other techniques. There are four values to calculate; the first and second are the number of occurrences of entities that only appear in concept set A and the number of occurrences of entities that only appear in concept set B. The second two values represent the number of *extra* occurrences of entities that appear in both concept set A and concept set B. The third value represents extra occurrences in partition A and the fourth value represents extra occurrences in partition B.

Because the MoJo values for different clustering methods will be compared to each other, as well as to statistics about the way the code itself changes from version to version, each value must be presented as a percentage of the entities considered by each clustering method for each version. For example, it is unreasonable to compare MoJo values for Part Type with values for concept sets because each set of concepts has roughly ten times the number of entities contained in a Part Type partition. Therefore, the MoJo values have been divided by the size of the partition the MoJo value was calculated on and multiplied by 100 to provide a percentage of that size.

5.5 Tool Support

Some tools were required to complete the analysis described in this chapter. The tools were mostly used to extract information from the Bauhaus Rigi results. The algorithm to

calculate the MoJo distance metric was also implemented, both in its original form, described in Section 4.3.3, and with the alterations described in Section 5.4. Because a large amount of the work involved in the analysis was text manipulation, these tools were implemented using Perl.

Bauhaus Rigi, the software suite described in Section 4.3 that was used to execute the chosen software clustering and concept analysis techniques, can output its representations as text files. All representations are presented as a graph with entities as nodes and connections between entities as edges. The text file produced contains a list of node facts, one for each entity in the graph, followed by a list of edge facts, one for each of the connections between the nodes. Each node fact contains the name, type and location (the file the node belongs to) of the node it represents. Each edge fact contains the type of the edge it represents and the names of the two nodes the edge connects.

A representation of the basic software system is used as input for the software clustering and concept analysis techniques. The output produced by each technique is a representation where new nodes are created, one for each cluster or concept, and the only edges are ones connecting entities to these cluster or concept nodes. Sixteen such representations were produced by running each of the four techniques described in Section 5.2 on each of the four versions described in Section 5.3.1. Figure 5.1 shows the use of Bauhaus Rigi for a single version of the software system.

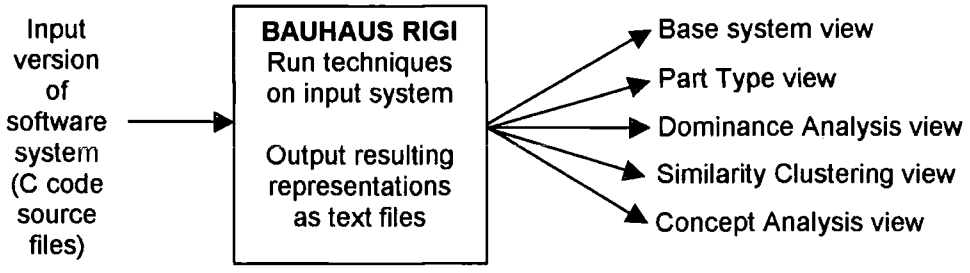


Figure 5.1 Use of Bauhaus Rigi

A tool called *count* was developed to count the number of each type of entities and connections there were in each representation. The tool calculates the number of lines containing the name of each type of entity and connection. For the entity types, the tool also records the location of the entity to determine whether the entity was part of the test component, part of the whole software system or contained in the standard libraries. This tool was used to generate the results contained in Figure 6.1 and Tables 6.1 to 6.4.

The MoJo algorithm, described in Section 4.3.3, and the adaptations described in Section 5.4 were implemented from scratch for this thesis. Because the MoJo metric is only concerned with the system itself, a basic *filter* tool was written to remove the library and system entities and all connections to these entities from the representations.

Two tools were created to implement the MoJo algorithm. The first tool was used for Similarity Clustering, Part Type and Dominance Analysis and implements both the basic algorithm described in Section 4.3.3 and the adaptations necessary to deal with different sets of entities described in Section 5.4. The second tool was used for Concept Analysis and implements the MoJo calculation method for Concept Analysis described in Section 5.4. These implementations generated the results shown in Tables 6.3, 6.4 and 6.5.

Finally, a tool called *printgroup* is used to print out the sets of clusters or concepts from each representation so that these sets could be analysed for the case study described in Section 5.3.2. This tool was used to assist the generation of Tables 6.7 to 6.10.

The use of the tools described here on each system view to create the data in Chapters Five and Six is shown in Figure 5.2. *MoJo* in this figure represents both the MoJo implementation for software clustering and the implementation for concept analysis.

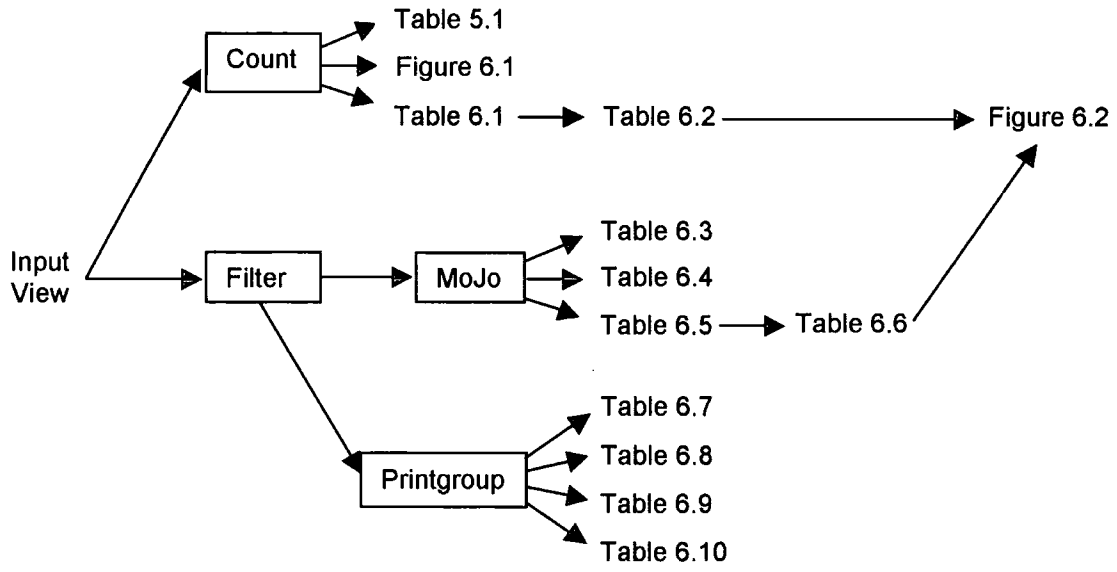


Figure 5.2: Tool support for analysis

5.6 Conclusion

This chapter has explained the analysis performed to assess whether software clustering and concept analysis techniques may prove useful for the purpose of studying software evolution. The methods for analysing the coverage and stability of these techniques have been described and the case study performed on a single group of functions has been outlined. The tools used to generate the results of this analysis have also been described. The following chapter explains the results achieved by these methods and case study.

CHAPTER SIX - Results

This chapter details and analyses the results acquired after the work described in Chapter Five was completed. The performance of the software clustering and concept analysis techniques that were tested is explored in detail in terms of the coverage and stability of each technique. A discussion of the case study outlined in Section 5.3.2 is also included.

6.1 Coverage

As explained in Section 5.1, it is important that the representations provided by the software clustering and concept analysis techniques examined cover as much of the system as possible, in order to allow a maintainer of the system to make an informed decision. This section demonstrates the coverage achieved by using Similarity Clustering, Dominance Analysis, Concept Analysis and Part Type to create views of the software system described in Section 5.3.

The graphs contained in Figure 6.1 show the percentage of functions, variables, types, system entities and library entities contained in the representations created by the four techniques. It may help to refer to Table 5.1 in Section 5.3.1 to put these percentages in perspective (definitions of system entities and library entities can also be found in that section).

It is immediately apparent that some techniques achieve much better coverage than others. However, a few of the lower levels of coverage are due to other reasons than an actual failure of the technique. These reasons will become clear as each graph is discussed.

Similarity Clustering clearly provides the best overall coverage. The noticeably lower percentage of coverage for System Entities in Version One is due to that fact that there are only six System Entities used by that version, of which Similarity Clustering covers four; the remaining versions have ten times that number of System Entities and Similarity Clustering covers those to a more than satisfactory level. Similarity Clustering covers all of the functions in all four versions. However, it should be noted

that this coverage is only representative of a single partition of the dendrogram produced by Similarity Clustering, and as this partition was taken from the latter stages of the clustering process a reasonable level of coverage should be expected.

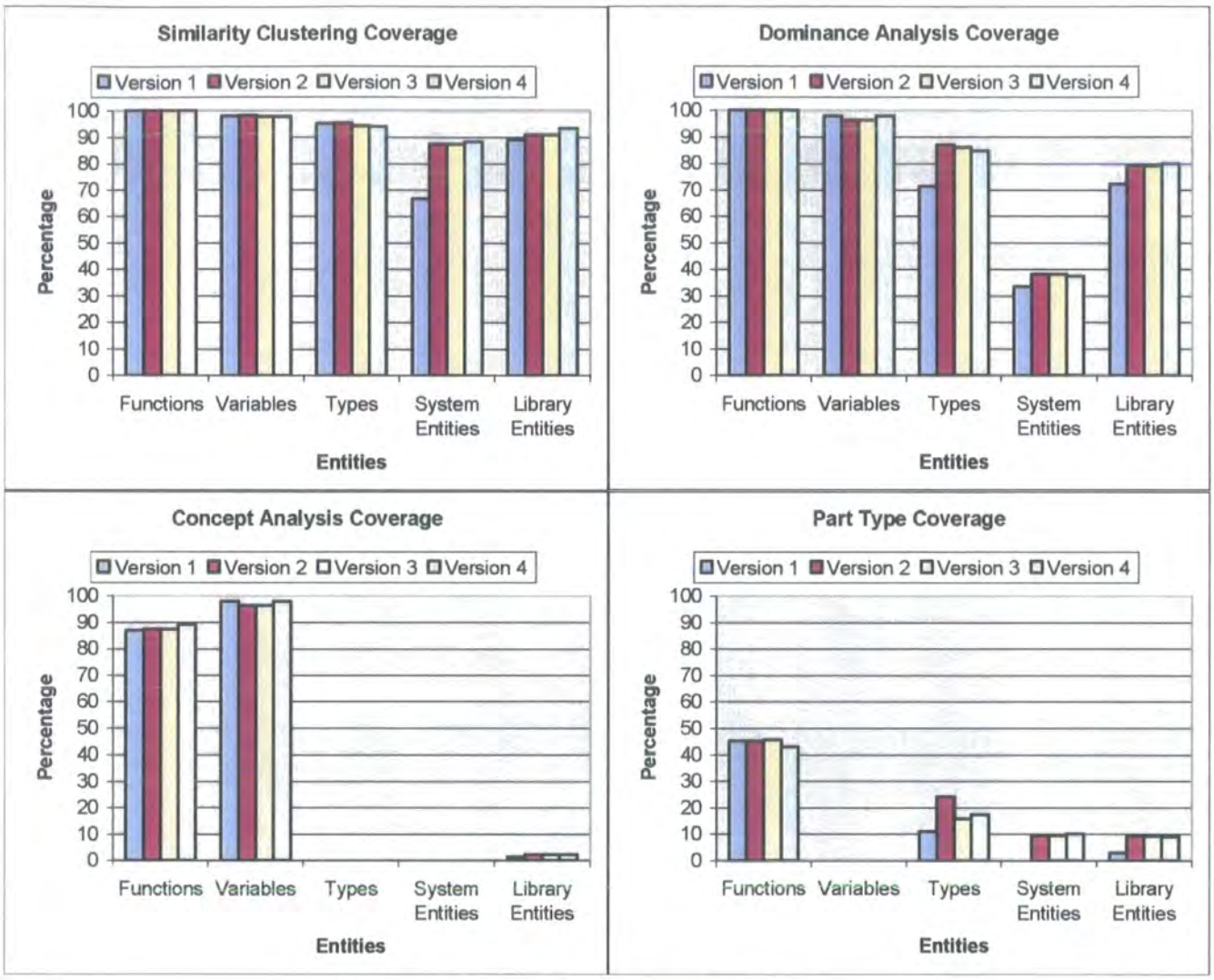


Figure 6.1: Coverage graphs

Dominance Analysis also provides good overall coverage. It suffers slightly when considering types; the poor coverage of System Entities might result from the fact that almost all of the System Entities used are types. However, the results shown here are quite misleading; in Koschke's implementation of Dominance Analysis, when an entity is not dominated by any other entity it is placed at the top of the dominance tree. This means the entity has strictly been covered, but has not actually been clustered with any other entities, which might explain the high levels of coverage shown here.

The absence of coverage for types (and therefore System Entities) by Concept Analysis is because the binary relation used as input for the Concept Analyser was based on

functions and variables alone. It would be easy to provide a relation for functions and types; however, because of the nature of a binary relation, it would be slightly harder to create suitable input including functions, variables and types. Nevertheless, the graph shows the Concept Analysis technique's decent coverage of functions and variables and there is no reason why using types would not produce similar levels of coverage.

Part Type provides the only dissatisfactory coverage. Unlike Concept Analysis and types, Part Type's failure to cover variables is a property of the technique itself, rather than its input. It might be possible to accept this if the coverage for other types of entity was better. Unfortunately, as shown by the graph, there is extremely poor coverage of both functions and types and practically no coverage of outside entities. This information means that Part Type is of no real use on its own; however, it may have a use as a means of confirming or questioning clusters presented by other techniques.

6.2 Stability

Section 5.1 outlines why it is important that software clustering and concept analysis should be as stable as possible if they are used to study the evolution of a software system. The MoJo distance metric developed by Tzerpos and Holt and described in Sections 4.2.3 and 5.4 has been used to test the stability of the four techniques tested. MoJo values have been calculated between Versions One and Two, Versions Two and Three and Versions Three and Four for each of the four techniques.

Tzerpos and Holt's work, described in Section 4.2.2, suggests that the MoJo values calculated here, when presented as a percentage of the total size of the versions the values have been calculated on, should be slightly less than, or at least match, the percentage change of the system's attributes from one version to the next if the technique being tested is to be considered stable. This would mean a technique has taken the changes in the system into account without drastically altering the representation it produces, thus aiding the comprehension of the representations.

	Version One	Version Two	Version Three	Version Four
Entities	411	431	436	488
Connections	2535	3191	3183	3640

Table 6.1: Entities and connections belonging to software system

Table 6.1 shows the number of entities and connections in each version of the software system used to test the techniques. A breakdown of the number of different types of entities can be seen in Table 5.1. The number of connections is the sum of various types of connection between entities in the system, such as the uses of a variable or the call of a function.

	V1 -> V2	V2 -> V3	V3 -> V4
Entities	4.87	1.16	11.93
Connections	25.88	0.25	14.36

Table 6.2: Percentage change of entities and connections

Table 6.2 shows the percentage change in number of entities and connections from one version to the next, based on the figures in Table 6.1. These figures roughly show the development of the system. It can be seen that there was an increase in complexity from Version One to Version Two, with the increase in the number of connections being over five times larger than the increase in the number of entities. There was only a small increase in size and complexity between Versions Two and Three. The effects of the preventative maintenance between Versions Three and Four can be seen in this table, where, although there has been a large increase in the number of entities, the increase in the number of connections is less than between Versions One and Two and more in line with the increase in the number of entities.

	Version One	Version Two	Version Three	Version Four
Similarity	402	463	466	513
Part Type	62	74	76	79
Dominance	376	430	434	484
Concept	899	1109	1132	1354

Table 6.3: Version sizes for MoJo calculation

Table 6.3 shows the size of each representation used to calculate MoJo values. Note that the concept sets are substantially bigger than the other representations because they can include multiple instances of the same entity.

	A=V1, B=V2		A=V2, B=V3		A=V3, B=V4	
	In A not B	In B not A	In A not B	In B not A	In A not B	In B not A
Similarity	79	140	2	5	41	88
Part Type	1	13	0	2	3	6
Dominance	59	113	0	4	35	85
Concept Extra	82	250	52	86	48	192
Concept Only	43	169	0	6	193	343

Table 6.4: Entities omitted for MoJo calculation

As reported in Section 5.4, MoJo can only be calculated on entities that exist in both of the representations being considered. Table 6.4 shows the number of entities that only existed in one version but not in the other for each pair of versions. Once again, the sets of concepts are special cases; Concept Only is the number of entities that only appear in one version but not the other, whereas Concept Extra is the number of extra instances of entities that exist in both versions. It should be noted that these are actual figures, not percentages; they should be considered in the context of the version sizes shown in Table 6.3.

	A=V1, B=V2		A=V2, B=V3		A=V3, B=V4	
	mno(A,B)	mno(B,A)	mno(A,B)	mno(B,A)	mno(A,B)	mno(B,A)
Similarity	103	**84**	**24**	37	**65**	71
Part Type	**2**	21	**0**	0	**0**	0
Dominance	18	**17**	**2**	2	**9**	10
Concept	**102**	108	**63**	83	**63**	73

Table 6.5: MoJo results (MoJo(A,B) highlighted)

Table 6.5 shows the actual MoJo values calculated by the heuristic MoJo algorithm. As described in Section 4.3.3, the MoJo value for two representations A and B is the

minimum of the two values mno(A,B) and mno(B,A). In Table 6.5, the minimum of the two values for each pair of values is highlighted, showing each of the values of MoJo.

	V1 and V2	V2 and V3	V3 and V4
Similarity	19.42	5.17	13.28
Part Type	2.94	0	0
Dominance	4.22	0.46	1.96
Concept	10.16	5.62	5.07

Table 6.6: MoJo results as a percentage of version size

In order to compare the MoJo values in Table 6.5 to each other and to the information in Table 6.2, they must be related to the size of the versions they were calculated for. The results in Table 6.6 are the MoJo values from Table 6.5 as a percentage of the average of the size of the two versions they were based on.

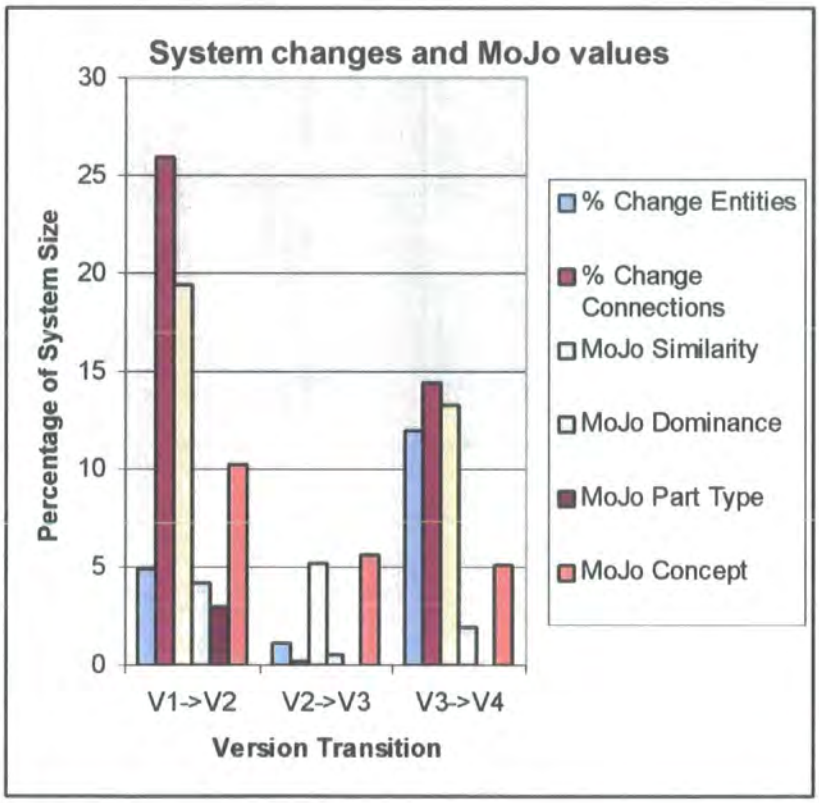


Figure 6.2: System changes and MoJo values as percentage of system size

Figure 6.2 contains both the MoJo results shown in Table 6.6 and the system statistics from Table 6.2. It should be noted that the discrepancies in this graph are not as severe as they may first appear, because the values given are percentages but the scale of the

graph only allows for figures of up to thirty percent in order to make the smaller values clear.

The graph in Figure 6.2 shows that all techniques are stable when compared to the percentage change of connections. However, this is not necessarily desirable if the techniques ignore significant changes in the system. For example, Part Type's representation of shared entities does not change from Version Two to Version Four, despite fifteen percent of the system's connections changing. While the representation is stable, the aim is for techniques to provide stable representations that take changes to the system into account. Inevitably there is a trade-off between stability and inclusion of changes, but taking no changes into account makes the representations almost meaningless.

The representations produced by Dominance Analysis suffer from the same problem to a lesser extent. However, as discussed in Section 6.1, the version sizes for Dominance Analysis are unfair because entities are included that have not necessarily been clustered. Therefore, the actual percentage change may be higher than displayed here and the results may be more useful than they appear.

The concept sets produced by Concept Analysis seem to be stable enough to allow useful comparison for maintenance purposes. While the MoJo values do not approach the percentage of connection changes, they are sufficiently large to suggest that these changes have been taken into account to some extent but the concept sets have still remained similar enough to allow easy comparison. The only undesirable result is the high MoJo value for the transition from Version Two to Version Three. This value suggests that Concept Analysis can be affected greatly by small changes and may imply that these changes are more important than they actually are.

The results for Similarity Clustering also suffer from this problem, which is usually encountered with hierarchical agglomerative clustering algorithms. Because of the sequential nature of the process, early decisions inevitably affect the rest of the decisions made. Therefore, if decisions are made differently for two different versions early in the clustering process, the entire dendrograms for these systems will be altered. This may also contribute to the high MoJo values for the transitions from Versions One to Two and Versions Three to Four. However, the values are still lower than the

percentage change of connections, suggesting that Similarity Clustering is the technique most likely to take important changes into account while still producing fairly stable representations.

6.3 Case Study

This section describes how the four software clustering and concept analysis techniques tested treat the group of functions described in Section 5.3.2. Each function in the group was searched for throughout the representations created by the techniques and the clusters or concepts containing any of the functions were recorded. The tables below show the size of each recorded cluster and the number of functions from the group that are also in each cluster for each of the four versions.

It should be reiterated here that, although the authors of the software system defined this group of functions, this does not guarantee that this group does in fact encapsulate an element of functionality. The fact that the group was not changed over four versions of the system suggests a lack of document maintenance more than a lack of changes to the use of the group. If techniques do not pick out the group successfully, that does not necessarily mean they are incorrect; it may mean they have uncovered some extra connections between the group of functions and the rest of the system, which is precisely why a maintainer may want to use the techniques in the first place.

6.3.1 Part Type Case Study

PART TYPE	Cluster One		Cluster Two	
	Group	Size	Group	Size
Version One	2	17	10	12
Version Two	2	47	10	12
Version Three	2	47	10	12
Version Four	2	47	10	12

Table 6.7: Part Type Case Study results

Table 6.7 shows where functions from the case study group are found in Part Type's representations. For example, in Version One, they appear in two clusters. The first

cluster contains two of the group's functions as part of a total of seventeen entities; the second cluster has twelve entities, ten of which are in the case study group.

It is clear that this second cluster effectively represents the group as a whole. Part Type has picked out this group because it centres on the data type DATABASEFILE. The two functions not in this group are *calcchecksum* and *datafilecreate*, which Part Type has chosen to cluster with a similar type, DATAINDEXFILE.

It can also clearly be seen how static the Part Type representation is, as already suggested in Section 6.2. The two clusters do not change at all from Version Two to Version Four. This may be partly because Part Type does not consider variables or variable usage but even based on functions and types alone some amount of change would be expected. The fact that Part Type has identified the case study group of functions suggests that the group must be very cohesive, given Part Type's poor coverage. This may mean that Part Type can be used to check the results provided by other software clustering and concept analysis techniques, but the fact that it fails to consider large amounts of the information available means that relying on Part Type alone is not recommended.

6.3.2 Dominance Analysis Case Study

The results for Dominance Analysis, shown in Table 6.8, are even less promising than the results for Part Type. As mentioned elsewhere, all entities that are not dominated by any other entity or are part of a cycle are placed in a single cluster. The results show that at least ten of the functions in the case study group form part of this cluster (Cluster One in Table 6.8), which contains a vast majority of all the entities in the system.

DOMINANCE	Cluster One		Cluster Two		Cluster Three	
	Group	Size	Group	Size	Group	Size
Version One	11	273	1	19		
Version Two	10	314	1	22	1	2
Version Three	10	316	1	22	1	2
Version Four	10	356	1	25	1	2

Table 6.8: Dominance Analysis Case Study results

This is not surprising because Dominance Analysis clusters functions with a common functional purpose, rather than ones with a data structure in common. For example, *recordadd* is included in Cluster Two, which contains a group of functions that are used to process requests sent to the system, which often leads to the adding of a record to the database. *calcchecksum* is the function contained in Cluster Three.

It is perhaps a little unfair to test Dominance Analysis against this group of functions because of these crossed purposes. There is still a possibility that Dominance Analysis could be useful if the maintainers were attempting to change or add to the functionality of the system, rather than examining the use of a data type or variable.

6.3.3 Similarity Clustering Case Study

SIMILARITY	Cluster One		Cluster Two		Cluster Three	
	Group	Size	Group	Size	Group	Size
Version One	9	129	3	76		
Version Two	3	162	8	14	1	14
Version Three	11	176	1	14		
Version Four	3	158	8	24	1	42

Table 6.9: Similarity Clustering Case Study results

At first glance, the results for Similarity Clustering, shown in Table 6.9, appear to demonstrate the same problem as the Dominance Analysis results, with Cluster One seemingly being too big for any conclusions about its contents to be accurate. However, while the size of this cluster is large, it should be remembered that the entities contained in it are functions, variables and types, and that when it is broken down into these three parts it may become easier to comprehend.

The fact that in Version One and Version Three the majority of the case study group is found in this large cluster demonstrates one of the problems with partitioning a dendrogram. Ideally, it would be possible to present a single representation of the system by taking a single partition from the dendrogram, as has been done here. However, from examinations of the rest of the dendrograms produced, the case study group was actually clustered together very early on in the clustering process and then clustered with many other groups of functions.

This suggests that, as mentioned earlier when discussing Part Type's case study results, the case study group of functions is highly cohesive. Unfortunately, this cannot be seen from the single partition taken from the dendrogram. It is possible that an earlier partition could be taken which would show this cohesion but this would mean the exclusion of the entities that had not been clustered at that point of the process.

The instability of Similarity Clustering can also be seen from Table 6.9. In particular, the second clusters for Versions Two and Four (which contain eight of the case study group's functions) do not feature in Version Three; this cluster has been joined with the large cluster (Cluster One). It is worth noting that the large cluster for Version Three is fourteen entities larger than the large cluster for Version Two, exactly the size of Cluster Two in Version Two, suggesting that this is the only change to the large cluster and that this change been caused by a small change between Versions Two and Three, highlighting the sensitivity of Similarity Clustering.

However, Cluster Two in Versions Two and Four does contain the majority of the case study group, despite the merging of Clusters One and Two in Version Three. The three entities in Cluster One for Versions Two and Four are *recordcheck*, *datafileclose* and *datafilecreate*. Once again, the rogue function found in Cluster Three for Versions Two and Four and Cluster Two for Version Three is *calcchecksum*.

6.3.4 Concept Analysis Case Study

The Concept Analysis results, shown in Table 6.10 are considerably more complex than the results for the other techniques. The concept lattices produced by Concept Analysis proved to be too complicated to partition automatically and so the actual set of concepts must be used. This means that the entities from the case study group can appear in more than once concept, which means many more concepts are affected than for other techniques. Therefore, some abbreviations have been used to keep the table to an acceptable size. V1 to V4 represent Versions One to Four, C1 to C9 represent Concepts One to Nine, G represents Group and S represents Size.

Concept Analysis isolates the case study group quite accurately; Concept One, with nine group functions in a cluster of thirteen entities, remains constant throughout the four versions. The three functions not featured in this cluster are *recordadd*, *datafileclose*

and *calcchecksum*. The one or two entities featured in the rest of the clusters are usually the data file functions, particularly *datafilecreate*, grouped with other file manipulation functions not in the case study group.

	C1		C2		C3		C4		C5		C6		C7		C8		C9	
	G	S	G	S	G	S	G	S	G	S	G	S	G	S	G	S	G	S
V1	9	13	2	37	3	29	1	36	1	25	1	22	1	13	1	5		
V2	9	13	2	46	2	42	1	40	1	25	1	26	1	7				
V3	9	13	2	154	2	42	1	61	1	40	1	20	1	19	1	15	1	7
V4	9	13	2	167	2	59	1	58	1	41	1	23	1	18	1	7		

Table 6.10: Concept Analysis Case Study results

These scattered references suggest a problem with the use of Concept Analysis. Here, where a group of functions is being considered, it is fairly easy to pick out Concept One as most representative of the group and trace the evolution of that group by following that concept. However, if only a single function was being searched for, it would be very difficult to understand how that function affected the system and which entities have an impact on that function if the function appeared in a large number of sizable concepts.

It would be possible to assess the general pattern of use for this function; for example, if it only appeared in one concept for Version One but ten for Version Four, then it is almost certain that the usage of this function and its interconnectivity with the rest of the system has increased dramatically. However, it is unlikely that any more specific information could be gleaned from a set of concepts without comparison with other system representations.

6.3.5 Case Study Conclusions

By examining the representations of the four versions created by the four techniques, a number of conclusions about the code can be drawn. Firstly, the case study group is highly cohesive, because much of it has been isolated by at least three of the techniques. Secondly, *calcchecksum* has not appeared with the other functions in the group in any representation, suggesting that this function does not belong in the case study group and was included erroneously by the authors of the system.

Thirdly, and most importantly in the context of evolution, while the case study group has remained cohesive the use of this group has increased over the four versions; this can be seen by the increase in the size of the clusters the group is found in, and the increase in the number of clusters or concepts functions in the group appear in. Therefore, it could be suggested that a useful preventative maintenance task would be to try and encapsulate this group a little more so that if a change needed to be made to one of the functions in the group it would affect as little of the rest of the system as possible.

It should be noted that it would be impossible to assert the observations above by examining only the results of a single technique; none of the techniques are accurate enough on their own to be certain of the results they produce. Therefore, it is strongly recommended that if these techniques are to be used to study evolution (and the results in this section suggest that this could be done productively) more than one technique should be used and the representations they produce compared before any conclusions are drawn.

6.4 Conclusion

This chapter has described the results achieved by carrying out the work outlined in Chapter Five. The coverage and stability of the four software clustering and concept analysis techniques has been tested and a number of observations have been made based on the representation of the case study function group described in Section 5.3.2. The following chapter draws some conclusions based on these results.

CHAPTER SEVEN - Conclusions

This final chapter summarises the work described in the rest of this thesis. The thesis has examined the possibility of using software clustering and concept analysis techniques to examine the evolution of software. The literature survey has reviewed a wide range of software clustering and concept analysis techniques and general work on the techniques. An analysis concerning the possibility of using software clustering and concept analysis techniques to study the evolution of software has been carried out on an industrial software system. The analysis contained some experimental analysis and a case study. Various small tools were developed to support this analysis and collect the results that were described in the previous chapter.

The following conclusions are based on the criteria for success listed in Section 1.5. Firstly, the history of software clustering and concept analysis that informed the work is outlined in Section 7.1. The work performed and the results of this work are explained in Sections 7.2 and 7.3. A summary discussion of the criteria for success is provided in Section 7.4. Finally, in Section 7.5, some recommendations for further work are made.

7.1 History

There have been a great number of software clustering and concept analysis techniques developed since the early 1980s. These techniques attempt to represent a software system as groups of similar or connected entities, in the hope that these representations will provide a better understanding of the software system. The field has developed from and been based on the existing multi-discipline field of cluster analysis. Over the years, researchers have attempted to use these techniques for a wide range of purposes, from program comprehension to reengineering to reuse. To satisfy Criteria 1, a full discussion of the development of the field can be found in Chapters Two and Three.

In recent years various researchers have attempted to summarise and classify the existing software clustering and concept analysis techniques. This work is explored in Chapter Four. One of the most thorough approaches is by the Bauhaus team, particularly through the work of Rainer Koschke. Koschke provides a succinct

taxonomy of these techniques in his doctoral thesis [KOSC00a], proposing four main categories of such techniques: connection-based, metric-based, graph-based and concept-based.

Currently, software clustering and concept analysis techniques do not provide the robustness or completeness required if the representations they produce are to be used to reengineer a system or propose reuse candidates. Perhaps unsurprisingly, most techniques can cluster the cohesive elements of a software system well but struggle to cope with entities that are very frequently used or dissimilar to the rest of the system's entities. However, it is possible that this property could be advantageous for purposes that do not require full representations of a software system.

This thesis has attempted a preliminary exploration into whether these techniques can be used to study the evolution of a software system. It is possible that by analysing suitable representations of different versions of a system, evolutionary trends could be uncovered and undesirable developments could be halted by applying preventative maintenance. This kind of application is likely to focus on a small group of entities rather than the whole system and so the representations produced by software clustering and concept analysis techniques may be useful for this purpose. As demanded by Criteria 5, the work described in Section 7.2 attempted to establish if this theory is worth pursuing.

7.2 Work Performed

Four techniques were tested, one for each of the categories established by Koschke [KOSC00a]. These were Part Type (connection-based), Similarity Clustering (metric-based), Dominance Analysis (graph-based) and an implementation of Concept Analysis by Lindig [LIND97] based on functions and variables. The clustering suite Bauhaus Rigi was used to run the techniques. Each technique was executed using an industrial software system as input, as demanded by Criteria 3. This software system has four versions and representations were created for each version by each technique.

These representations were then assessed to establish the coverage and stability of each technique, as stated in Criteria 2. Coverage is the amount of entities that were

considered by the technique during the clustering process. It is important that a high degree of coverage is attained in order to validate the representations produced; if a large number of entities is omitted and does not inform the clustering process any observations about the representations are likely to be flawed.

Each technique must also be stable; this means they must change in line with changes from version to version. Some clustering techniques are prone to alter wildly with only a slight change in the input system. The stability of the assessed techniques was tested using a distance metric called MoJo, developed by Tzerpos and Holt [TZER99]. A heuristic algorithm used to calculate MoJo was implemented for this thesis; this implementation adapted the algorithm to take into account two inputs of different sizes and multiple entities of the same name. This implementation and other tools that were developed to aid the manipulation of results produced by Bauhaus Rigi satisfy Criteria 4.

A case study was also completed for this thesis. A group of functions known to be considered cohesive by the authors of the tested software system was traced throughout the representations created by the assessed techniques. This study was undertaken in the hope that some observations could be made about the nature of the techniques and the software system, which in turn may lead to some general observations about the possibility of studying software evolution using the techniques. Further information on the work performed for this thesis can be found in Chapter Five.

7.3 Results Analysis

The results of testing the four techniques Part Type, Dominance Analysis, Concept Analysis and Similarity Clustering are now discussed. The analysis for each technique, detailed in Chapter Six, is explained and some more general conclusions on the use of software clustering and concept analysis techniques for the study of software evolution can then be drawn.

Part Type, the connection-based approach developed by Liu and Wilde [LIU90] and explained in Section 3.1.4, is one of the earlier software clustering approaches. It is typical of connection-based approaches (and, in fact, many early software clustering

approaches) in that it only considers a single set of relationships in the system, in this case the use of types by functions.

The results show that Part Type produces reasonable clusters based on the information it uses and that it can pick out the important functions related to a type. However, the information base used by Part Type is extremely small; only types that are directly used by functions are included. If a function uses a variable of a certain type but does not use the type itself, Part Type will include neither the function nor the type in its representation of the system.

As shown in the coverage analysis of Part Type, for the test system, this meant that less than half of the functions and under a quarter of the types in the system were included for all four versions. As Part Type does not include any variables either, this makes it very difficult to use the results with any degree of certainty. This observation is validated because Part Type's representation remains the same (completely stable) from Version Two to Version Four, despite large changes to the system.

Unfortunately, these problems are likely to affect most connection-based approaches because they usually focus on one type of connection alone. Also, basing the results on a number of different connections is likely to cause a number of conflicts because each connection will generally produce different clusters. However, this is not to say that connection-based approaches are completely useless. It is thought that they can be very useful as a confirmation tool, to validate a clustering suggested by another technique. This is particularly appropriate because connection-based approaches are typically very fast and easy to execute, due to their natural simplicity.

Dominance Analysis, the graph-based approach developed by Cimitile and Visaggio [CIMI95] and described in Section 3.1.10, seems to have good coverage and stability from the results in Chapter Six. However, as explained elsewhere, because Dominance Analysis is based a specific relation it fails to cluster a large number of entities in the system, although it does include these entities in its representation. The dominance relations it does define tend to only concern one or two entities, with only a few groups containing a sizeable number of entities.

The case study shows that this makes it very difficult to draw any reasonable conclusions from the representations Dominance Analysis produces, but also suggests that this is because Dominance Analysis is based on functional connections rather than data structures. It is also only likely to work with any success on well-structured systems because it picks out the local functions to other functions; if a function A is strongly dominated by a function B that means that only function B uses function A. A frequently altered system is likely to have few of these local functions left. However, this may mean that Dominance Analysis will be perfect for tracking the evolution of system where a well-defined, cohesive set of functions has dissipated over time.

The Concept Analysis approach used here is based on the work of Lindig [LIND97] and described in Section 3.2. It has produced promising results, despite creating sets of concepts that proved too difficult to partition. It covers the majority of the functions and variables in the system and is stable enough to take the representations it produces seriously. As mentioned in Chapter Six, the omission of types is a problem with the input to the analysis, not the analysis itself, which could be adapted to take into account any number of features.

Concept Analysis also managed to isolate the majority of the case study group consistently, while also suggesting how the use of the group may have changed over the development of the four versions of the test system. Unfortunately, it would take a reasonable amount of manual analysis by the maintainer of the set of concepts and the code itself to make these changes clear, because the set of concepts contains overlapping and sometimes contrasting juxtapositions of entities. Cross-examination with the results from Part Type, for example, may help to confirm observations drawn from the sets of concepts.

The most promising results were produced by Similarity Clustering, the metric-based approach designed by Koschke [KOSC00a] based on earlier work by Schwanke [SCHW91] and described in Section 3.1.13.4. This approach is the most thorough and complicated software clustering method available, which means that it takes some time to produce its representations. It has the best coverage of all tested techniques and has levels of stability closest to the actual change in the system, suggesting it takes the changes made from version to version into account while still retaining continuity in its representations.

Unfortunately, it is possible that Similarity Clustering, because it considers so much information, produces representations that may be too complicated to be understood easily. This is demonstrated by the changes in the representation of the case study group of functions, where clusters disappear and reappear again due to small changes that Similarity Clustering is perhaps over-sensitive to. There is also a major problem (as with all hierarchical agglomerative algorithms) with the partitioning of the dendrogram produced by Similarity Clustering; in fact, as with Concept Analysis, it is recommended that the actual dendrogram be studied for the purposes of evolution rather than partitioning the dendrogram. While this provides a large amount of information for the maintainers of a system to examine, with a clear aim in mind it should be easy to select the relevant parts of the dendrogram for each version and trace the development of the part of the system being studied.

Metric-based approaches such as Similarity Clustering are more versatile than other types of approaches, because any information about the system can be used as input provided that input can be represented numerically. The amount and importance of information in the input to the approach can be changed with ease without the user having to alter the rest of the clustering process. This is also true of Concept Analysis, but the binary relation used as input for Concept Analysis is less flexible than the metric used for Similarity Clustering.

In conclusion, and to provide the recommendations required by Criteria 6, while the results for Dominance Analysis and Part Type were disappointing, the representations produced by Concept Analysis and Similarity Clustering suggests that there is a use for these techniques to study software evolution. Changes to a group of functions can be traced using these approaches and the overall evolution of a software system can be examined from a number of different angles with ease. As many software systems have incomplete or non-existent documentation, even the statistical data about how the number of entities changes from version to version may prove useful to maintainers.

However, the use of these techniques can only be recommended if more than one technique is used, because at present the results from a single technique can not be trusted sufficiently to be relied on. This could prove to be a problem because of the nature of the maintenance of a software system. If these approaches were to be used to assess the impact of a corrective or adaptive maintenance change there is likely to be a

need for this change to be made very quickly. There may not be time to produce and compare a large number of representations. The same is true to a lesser extent of preventative and perfective maintenance.

Although more accurate and more immediate representations are needed, it is becoming easier and easier to produce these representations, even as the techniques themselves become increasingly complicated. This will only improve as further work is done into the uses of software clustering. As suggested above, if a maintainer has a clear aim in mind when using these techniques, the analysis of the representations produced may give the maintainer a different impression of the system or confirm already held suspicions about the system within a matter of minutes. Therefore, it is believed that these techniques could give the maintainer a significant advantage when further evolution of the system takes place.

7.4 Review of Success Criteria

The criteria for success stated in Section 1.3 have been referred to throughout this chapter. The following is a summary discussion of these criteria.

1. *To summarise the history of software clustering and concept analysis techniques.*

   Chapter Two discusses the overall development of the fields of software clustering and concept analysis. Many of the techniques that have been created during this development are explained in detail in Chapter Three. Chapter Four is concerned with work that has attempted to classify and evaluate these techniques. These chapters are summarised in Section 7.1.

2. *To assess the coverage and stability of a number of software clustering and concept analysis techniques.*

   Chapter Five describes the analysis performed for this thesis, including an assessment of coverage and stability. The methods of analysis are outlined in Section 5.1. The MoJo metric and algorithm used to calculate stability are described in Sections 4.2.2 and 4.2.3; some new adaptations to the MoJo algorithm are explained in Section 5.4. The results acquired during this analysis

are detailed and discussed in Sections 6.1 and 6.2. A small case study has also been performed to illuminate the coverage and stability results; this case study is described in Section 5.3.2 and the results of the case study are discussed in Section 6.3. The assessment is summarised in Section 7.2 and the results of the assessment are used to draw conclusions in Section 7.3.

3. *To use industrial strength software during the analysis of these techniques.*
   A single industrial software system served as input for the assessed software clustering and concept analysis techniques. This system is described in Section 5.3.

4. *To develop tools to support the analysis of these techniques.*
   A number of tools were developed to execute the MoJo algorithm and process the representations produced by the assessed software clustering and concept analysis techniques. The tools are described in Section 5.5 and the architecture formed by the tools and the other analysis software used during the assessment is visualised in Section 5.6.

5. *To investigate the feasibility of studying evolution of systems using these techniques.*
   Discussions throughout Chapter Six and in Section 7.3 form an investigation into the feasibility of studying evolution of systems using software clustering and concept analysis techniques. Based on these discussions, it is certainly possible to examine system evolution using these techniques. However, at present, this examination should be limited to small-scale assessment of single entities or groups of entities rather than whole system evolution. The techniques are not currently sufficiently developed to be relied upon but may prove useful as a clarification tool as part of some larger evolution analysis.

6. *To provide recommendations for the most appropriate techniques to use for the purposes of evolution.*
   Similarity Clustering provided the best overall results of the four techniques assessed for this thesis; Concept Analysis also provided good quality results. Part Type and Dominance Analysis provided poor results under the assessment criteria used for the analysis. However, the case study discussed in Section 6.3

shows that no results from a single technique can be relied upon and it is necessary to compare the results of different techniques to draw any observations. Therefore, a possible use for the techniques is to propose basic overall representations using Similarity Clustering or Concept Analysis and confirm connections between entities suggested by these representations using more specific connection-based techniques like Part Type. While this will mean more work for the maintainer, the techniques are very easy and quick to execute and so this extra comparison work should not prohibit the use of software clustering and concept analysis techniques for the study of evolution.

7.5 Further Work

This thesis has only undertaken a preliminary investigation of the use of software clustering and concept analysis techniques to study the evolution of software systems. This section outlines some of the further work that could lead from the investigation described in this thesis, both in the immediate future and afterward.

Only four software clustering and concept analysis techniques have been explored in this thesis. There are many more techniques that could have been assessed under the same evaluation criteria, including many of the techniques described in Chapter Three. Also, the techniques that have been analysed should be evaluated further; for example, the parameters for Similarity Clustering were not explored and these parameters can dramatically affect the representations that this technique produces.

The test software system used for this thesis is not particularly large and has only four versions; many software clustering techniques have difficulty coping with large systems and so an investigation on a grander scale could provide some interesting results. Also, examining a system that has intermediate versions between main releases, which would suggest smaller changes between versions, may make it easier to compare the representations produced by the assessed techniques.

It would be extremely valuable to execute a real-life survey of the use of these techniques. Actual system maintainers could be provided with clustering tools and encouraged to use the representations produced when maintaining their software, either

in a controlled environment with well-chosen example changes or in the workplace with whatever changes or maintenance tasks occur over a time period. This type of study could provide information not only on the ability of existing techniques but point to what maintainers may desire in a clustering method designed specifically to track the evolution of software.

The tools developed to support the analysis described here (listed in Section 5.5) are informal and would not be suitable for real maintenance use. Therefore, it would be useful to develop a software suite that would enable the application of software clustering and concept analysis techniques to study evolution for actual maintenance. This suite should focus on small-scale use of the techniques; for example, it should be designed to allow a maintainer to track the changes in use of a single entity or small group of entities efficiently without being concerned with a picture of the whole system. As maintenance teams are often under pressure to make changes to a system quickly and efficiently, the evolution software suite must be similarly quick and efficient and focussed on maintainers' needs if it is to be adopted for use by any real maintenance unit.

One of the major problems with software clustering and concept analysis techniques is that, at present, they only consider the syntactic elements of a system and cannot include any semantic, domain knowledge about a system. However, this is a problem with the input to the techniques rather than the techniques themselves. If it were made possible to suitably code this semantic information as descriptive features of entities, it could act as input to software clustering and concept analysis techniques, either alone or in conjunction with syntactic information.

With further development of software clustering and concept analysis techniques and further assessment of these techniques on many more software systems, it may be possible to use the techniques to study or propose more general theories of software evolution. As maintainers of systems can confirm or deny suspicions about particular software systems by examining suitable representations produced by software clustering and concept analysis techniques, software evolution researchers could assess more general observations about the evolution of software using these techniques. The results acquired over time may even suggest possible general trends of evolution that had not previously been considered.

7.6 Conclusion

This chapter has summarised the work carried out for this thesis and evaluated how successful this work has been based on the criteria for success outlined in Section 1.5. It is concluded that the work has been worthwhile because it has demonstrated that there is a very good case to be made for the use of software clustering and concept analysis techniques when studying the evolution of software systems.

REFERENCES

[ANDE73] Anderberg M.R., 'Cluster Analysis for Applications', Academic Press, ISBN 0-12-057650-3, 1973

[ANQU99] Anquetil N., Lethbridge T., 'Experiments with Clustering as a Software Remodularisation Method', Proceedings of the International Workshop on Program Comprehension, pp 235-255, 1999

[BELA82] Belady L.A., Evangelisti C.J., 'System partitioning and its measure', Journal of Systems and Software, 2(1):23-29, 1982

[BENN96] Bennett K., 'Software evolution: past, present and future', Information and Software Technology, 38:673-680, 1996

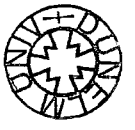
[BURD00] Burd E., Bradley S., Davey J., 'Studying the Process of Software Change: an analysis of software evolution', Proceedings of the Seventh IEEE Working Conference on Reverse Engineering, pp 232-239, 2000

[CANF96] Canfora G., Cimitile A., Munro M., 'An Improved Algorithm for Identifying Objects in Code', Software – Practice and Experience, 26(1):25-48, 1996

[CANF99] Canfora G., Cimitile A., De Lucia A., Di Lucca G.A., 'A Case Study of Applying an Eclectic Approach to Identify Objects in Code', Proceedings of the Seventh International Workshop on Program Comprehension, pp 136-143, 1999

[CANF00] Canfora G., Czeranski J., Koschke R., 'Revisiting the Delta IC Approach to Component Recovery', Proceedings of the Seventh Working Conference on Reverse Engineering, pp 140-149, 2000

[CHIK90] Chikofsky E.J., Cross J.H., 'Reverse Engineering and Design Recovery: A Taxonomy', IEEE Software, 7(1):13-17, 1990



[CHOI90] Choi S.C., Scacchi W., 'Extracting and Restructuring the Design of Large Systems', IEEE Software, 7(1):66-71, 1990

[CIMI95] Cimitile A., Visaggio G., 'Software Salvaging and the Call Dominance Tree', Journal of Systems and Software, 28:117-127, 1995

[DAVE00] Davey J., Burd E., 'Evaluating the Suitability of Data Clustering for Software Remodularisation', Proceedings of the Seventh Working Conference on Reverse Engineering, pp 268-276, 2000

[DEUR99a] van Deursen A., Klint P., Verhoef C., 'Research Issues in the Renovation of Legacy Systems', Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science volume 1577, pp 1-21, Springer-Verlag, 1999

[DEUR99b] van Deursen A., Kuipers T., 'Identifying Objects Using Cluster and Concept Analysis', Proceedings of the Twenty-first International Conference on Software Engineering, pp 246-255, 1999

[DOVA99] Doval D., Mancoridis S., Mitchell B.S., 'Automatic Clustering of Software Systems using a Genetic Algorithm', IEEE Proceedings of the International Conference on Software Tools & Engineering Practice, 1999

[EVER93] Everitt B., 'Cluster Analysis', Edward Arnold, ISBN 0-340-584793, 1993

[GIRA97] Girard J-F., Koschke R., Schied G., 'Comparison of Abstract Data Type and Abstract State Encapsulation Detection Techniques for Architectural Understanding', Proceedings of the Fourth Working Conference on Reverse Engineering, pp 66-75, 1997

[GIRA99] Girard J-F., Koschke R., 'A Metric-based Approach to Detect Abstract Data Types and State Encapsulations', Journal of Automated Software Engineering, 6(4):357-386, 1999

[GIRA00] Girard J-F., Koschke R., 'A comparison of abstract data types and objects recovery techniques', Science of Computer Programming, 36:149-181, 2000

[GOLD98] Gold N., 'The Meaning of "Legacy Systems"', Technical Report 7/98, Computer Science Department, University of Durham, 1998

[HUTC85] Hutchens D.H., Basili V.R., 'System Structure Analysis: Clustering with Data Bindings', IEEE Transactions on Software Engineering, 11(8):749-757, 1985

[JONE98] Jones C., 'The Year 2000 Software Problem', Addison-Wesley, ISBN 0-201-30964-5, 1998

[KAUF90] Kaufman L., Rouseeuw P.J., 'Finding Groups in Data: An Introduction to Cluster Analysis', Wiley Series In Probability And Mathematical Statistics, ISBN 0-471-87876-6, 1990

[KOSC00a] Koschke R., 'Atomic Architectural Component Recovery for Program Understanding and Evolution', Ph.D. Thesis, Institute for Computer Science, University of Stuttgart, 2000

[KOSC00b] Koschke R., Eisenbarth T., 'A Framework for Experimental Evaluation of Clustering Techniques', Proceedings of the Eighth International Workshop on Program Comprehension, pp 201-210, 2000

[LAKH97] Lakhotia A., 'A Unified Framework for Expressing Software Subsystem Classification Techniques" Journal of Systems and Software, 36:211-231, 1997

[LEHM97] Lehman M.M., Ramil J.F., Wernick P.D., Perry D.E., Turski W.M., 'Metrics and Laws of Software Evolution – The Nineties View', Proceedings of the Fourth International Symposium on Software Metrics, pp 20-32, 1997

[LIEN80] Lientz B.P., Swanson E.B., 'Software Maintenance Management', Addison-Wesley, ISBN 0-201-04205-3, 1980

[LIND97] Lindig C., Snelting G., 'Assessing Modular Structure of Legacy Code Based on Mathematical Concept Anlaysis', Proceedings of the Nineteenth International Conference on Software Engineering, p 349-359, 1997

[LIU90] Liu S.S., Wilde N., 'Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery', Proceedings of the International Conference on Software Maintenance, pp 266-271, 1990

[LIVA94] Livadas P.E., Johnson T., 'A New Approach to Finding Objects in Programs', Software Maintenance: Research and Practice, 6:249-260, 1994

[MANC98] Mancoridis S., Mitchell B.S., Rorres C., Chen Y., Gansner E.R., 'Using Automatic Clustering to Produce High-Level System Organizations of Source Code', Proceedings of the Sixth International Workshop on Program Comprehension, pp 45-53, 1998

[MANC99] Mancoridis S., Mitchell B.S., Chen Y., Gansner E.R., 'Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures', Proceedings of the International Conference on Software Maintenance, pp 50-59, 1999

[MEND97] Mendonça N.C., Kramer J., 'A Quality-Based Analysis of Architecture Recovery Environments', Proceedings of the First Euromicro Working Conference on Software Maintenance and Reengineering, pp 54-59, March 1997

[MÜLL93] Müller H.A., Orgun M.A., Tilley S.R., Uhl J.S., 'A Reverse Engineering Approach To Subsystem Structure Identification', Software Maintenance: Research And Practice, 5(4):181-204, 1993

[PARN72] Parnas D.L., 'On the Criteria To Be Used in Decomposing Systems into Modules', Communications of the ACM, 15(12):1053-1058, December 1972

[PATE92] Patel S., Chu W., Baxter R., 'A Measure For Composite Module Cohesion', Proceedings of the Fourteenth International Conference on Software Engineering, pp 38-48, 1992

[SAHR99] Sahraoui H.A., Lounis H., Melo W., Mili H., 'A Concept Formation Based Approach to Object Identification in Procedural Code', Automated Software Engineering 6(4):387-410, 1999

[SCHW91] Schwanke R.W., 'An Intelligent Tool For Re-engineering Software Modularity', Proceedings of the Thirteenth International Conference on Software Engineering, pp 83-93, 1991

[SCHW94] Schwanke R.W., Hanson S.J., 'Using Neural Networks to Modularise Software', Machine Learning 15:137-168, 1994

[SIFF97] Siff M., Reps T., 'Identifying Modules Via Concept Analysis', Proceedings of the International Conference on Software Maintenance, p170-179, 1997

[SNEA73] Sneath P.H.A., Sokal R.R., 'Numerical Taxonomy', W.H.Freeman & Sons, ISBN 0-7167-0697-0, 1973

[SNEL96] Snelting G., 'Reengineering of Configurations Based on Mathematical Concept Analysis', ACM Transactions on Software Engineering and Methodology 5(2):146-189, 1996

[TAKA96] Takang A.A., Grubb P.A., 'Software Maintenance: Concepts and Practice', International Thompson Computer Press, ISBN 1-85032-192-2, 1996

[TZER97] Tzerpos V., Holt R.C., 'The Orphan Adoption Problem in Architecture Maintenance', Proceedings of the Fourth Working Conference on Reverse Engineering, pp 76-82, 1997

[TZER98] Tzerpos V., Holt R.C., 'Software Botryology: Automatic Clustering of Software Systems', Proceedings of the International Workshop on Large-Scale Software Composition, pp 811-818, 1998

[TZER99] Tzerpos V., Holt R.C., 'MoJo: A Distance Metric for Software Clusterings', Proceedings of the Sixth Working Conference on Reverse Engineering, pp 187-193, 1999

[TZER00a] Tzerpos V., Holt R.C., 'On The Stability of Software Clustering Algorithms', Proceedings of the Eighth International Workshop on Program Comprehension, pp 211-218, 2000

[TZER00b] Tzerpos V., Holt R.C., 'ACDC: An Algorithm for Comprehension-Driven Clustering', Proceedings of the Seventh Working Conference on Reverse Engineering, pp 258-267, 2000

[WIGG97] Wiggerts T.A., 'Using Clustering Algorithms in Legacy Systems Remodularisation', Proceedings of the Fourth Working Conference on Reverse Engineering, pp 33-43, 1997

[YEH95] Yeh A.S., Harris D.R., Rubenstein H.B., 'Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language', Proceedings of the Fourth Working Conference on Reverse Engineering, pp227-236, 1997

[YOUR79] Yourdon E., Constantine L.L., 'Structured Design', Prentice Hall, ISBN 0-9170-7211-1, 1979